APR 01 2004

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	29.Mar.04	DISSERTATION

4. TITLE AND SUBTITLE
"EXTENDING THE REACH OF STATISTICAL SOFTWARE TESTING"

5. FUNDING NUMBERS

6. AUTHOR(S)
MAJ WEBER ROBERT J

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
UNIVERSITY OF MINNESOTA MINNEAPOLIS

8. PERFORMING ORGANIZATION REPORT NUMBER
CI04-137

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
THE DEPARTMENT OF THE AIR FORCE
AFIT/CIA, BLDG 125
2950 P STREET
WPAFB OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Unlimited distribution
In Accordance With AFI 35-205/AFIT Sup 1

DISTRIBUTION STATEMENT A
Approved for Public Release
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

20040406 014

14. SUBJECT TERMS	15. NUMBER OF PAGES
	167
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Dedication

To my wife, [redacted], without whom this would not have been possible.

To my sons, [redacted], [redacted], and [redacted], who had to put up with seeing less of Dad than they would have liked.

I can do all things through Christ who strengthens me. Phil 4:13

Acknowledgments

Dr. Mats Heimdahl, as my advisor, thanks for helping me to get this done. Drs. Baxter, Karypis, and Van Wyk, thanks for agreeing to be on my committee. I value your time, knowledge, and assistance.

Mike Whalen and George Devaraj, for your invaluable assistance with C++ and the inner workings of NIMBUS. What a treat to work with you guys!

Sherman Eagles of Medtronic, Inc. for bringing statistical testing to our attention, and Dr. Yunja Choi, for her early work with statistical testing at the U of M.

The United States Air Force Institute of Technology (AFIT) for giving me the opportunity to do this research. I hope I have proved myself worthy of the faith you have shown in me.

My boys, [redacted], [redacted], and [redacted], who have seen me in some form of school their entire lives. Thanks for putting up with the unusual hours.

[redacted], love of my life. What more can I say? You have been there through two Master's Degrees, a Ph.D., and more Air Force training than can be imagined. The debt I owe all the people above could not begin to compare to how much I owe you. Thanks for always being there and "cracking the whip" to make me focus on my work.

To God, the Father, Son, and Spirit - Oh, how you have blessed me so abundantly! Thank you for guiding me to this point in my life and strengthening me throughout this whole process.

THE VIEWS EXPRESSED IN THIS ARTICLE ARE THOSE OF THE AUTHOR AND DO NOT REFLECT THE OFFICIAL POLICY OR POSITION OF THE UNITED STATES AIR FORCE, DEPARTMENT OF DEFENSE, OR THE U.S. GOVERNMENT

Extending the Reach Of Statistical Software Testing

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Robert John Weber

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Dr. Mats P.E. Heimdahl, Advisor
February, 2004

Abstract

Statistical software testing is a promising technique for reducing the software testing burden by applying test cases to the software under test according to a model of the expected usage of the system in operation, called an operational profile. However, current statistical testing techniques have not been put into general practice, as they do not scale well as the complexity of the systems under test increases. In particular, as system complexity increases, the matrices required to generate test cases and perform model analysis can grow dramatically, even exponentially, overwhelming the test generation and analysis applications.

In this work, a new technique for representing the operational profile is proposed to mitigate the complexity issue for statistical test case generation. The use of a state-based requirements specification model as the basis of the operational profile leverages parallelism to reduce the visible size of the model. The specification model can then be extended to include conditional probabilities of input data occurrence. This work also proposes a statistical testing framework using this specification-based operational profile to generate and execute test cases. Such a framework allows us to evaluate the effectiveness and efficiency of this statistical testing technique.

This dissertation makes three key contributions. First, we have developed a technique to extend a specification to permit the creation of operational profiles with parallelism, reducing the size of the operational profile. Second, we have created a testing framework to take advantage of this new operational profile to statistically generate valid and effective test cases. Finally, we present a case study to show that our testing framework scales well and is capable of generating test cases that perform

comparably with test cases generated to provide specific levels of structural model coverage, given the same amount of effort (measured as time expended).

Abstract

Statistical software testing is a promising technique for reducing the software testing burden by applying test cases to the software under test according to a model of the expected usage of the system in operation, called an operational profile. However, current statistical testing techniques have not been put into general practice, as they do not scale well as the complexity of the systems under test increases. In particular, as system complexity increases, the matrices required to generate test cases and perform model analysis can grow dramatically, even exponentially, overwhelming the test generation and analysis applications.

In this work, a new technique for representing the operational profile is proposed to mitigate the complexity issue for statistical test case generation. The use of a state-based requirements specification model as the basis of the operational profile leverages parallelism to reduce the visible size of the model. The specification model can then be extended to include conditional probabilities of input data occurrence. This work also proposes a statistical testing framework using this specification-based operational profile to generate and execute test cases. Such a framework allows us to evaluate the effectiveness and efficiency of this statistical testing technique.

This dissertation makes three key contributions. First, we have developed a technique to extend a specification to permit the creation of operational profiles with parallelism, reducing the size of the operational profile. Second, we have created a testing framework to take advantage of this new operational profile to statistically generate valid and effective test cases. Finally, we present a case study to show that our testing framework scales well and is capable of generating test cases that perform

comparably with test cases generated to provide specific levels of structural model coverage, given the same amount of effort (measured as time expended).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Testing of software is generally an expensive, time consuming, and inaccurate method to remove software defects; new and innovative methods are necessary to focus limited resources on testing efforts that will be the most efficient. It is well known that exhaustively testing software is, in general, infeasible; even adequately testing software generally takes an enormous number of test cases. Statistical testing is one promising area of study, as it claims to provide significant savings by reducing the number of test cases required to thoroughly test a piece of software by focusing testing on the portions of code most used by the consumer. By focusing the testing effort on the code most used, the defects most likely to be encountered by the end-user are also the ones most likely to be found and eliminated during testing [27, 30, 31, 33], leading to a perception of higher reliability. Statistical testing determines the sections of code to test based on a model of the software's usage. This *usage model* determines the the frequency and sequencing of the inputs to the software. The usage model is typically captured using a Markov-model [56, 57, 72, 73] since they can be subjected to well-understood analyses, for example, the usage models can be analyzed for expected reliability, mean number of test cases needed until a section of code is encountered, and average length of test case. Naturally, there are limitations to statistical testing. Most seriously, Butler and Finelli, in [5], have shown that statistical testing cannot achieve the high levels of reliability desirable for safety critical systems—it is simply infeasible to run enough test cases. In addition, as the usage models become larger,

performing any type of analysis on the Markov-models also becomes infeasible. Nevertheless, we can still obtain benefits from statistical testing, such as the ability to generate large numbers of test cases more quickly than other testing strategies. Unfortunately, in our experience, even *building* a usage model using existing techniques can be an infeasible task—to accurately capture the usage of even a simple piece of software we typically need very complex models. Current language support for building usage models, in our opinion, is inadequate and limited to flat state machines or state machines with a limited support for hierarchical constructs [1, 31, 72, 73]; furthermore, even these hierarchical models generally are reduced to flat models before analysis or test case generation. To make statistical testing feasible in complex systems, our ability to provide larger and more accurate usage models is of critical importance—this problem is the main focus of this dissertation.

To extend the reach of statistical testing, we must be able to build usage models that accurately reflect the intended usage of the software. To this effect, we propose to use modeling notations intended for software specification as a basis for usage modeling. These notations are designed to capture enormous state-spaces in a compact form, typically by allowing both parallelism and hierarchies in the models. By extending hierarchical/parallel state machines with constructs to capture the probabilistic behavior, we hope to enable a tester to build larger and more accurate usage models than previously possible, and in that way increase the reach and applicability of statistical testing techniques.

The Critical Systems Research Group at the University of Minnesota has developed the Requirements State Modeling Language without Events (RSML^{-e}) [19]; a state-based modeling language that will serve as the basis for our statistical testing and analysis modeling. A formal language, it accurately describes requirements and creates a state-based model of a system's behavior. This model is our target for the

development of usage models using Markov chains. By using a specification language such as RSML^{-e}, we also hope to address another perceived problem of model-based statistical testing; that it requires that "testers be able to program" [47]. RSML^{-e} was developed to be used by non-computer professionals [34] and we believe this will be useful when the language is used by testing professionals.

• In this dissertation, we provide an extension to RSML^{-e} that will allow persons to create large models in a compact and readable way. With this new capability, we have evaluated random testing strategies (statistical testing with a uniform usage profile) on some larger case studies. The **long-range goal** of this research is to decrease the expenses of software testing. Augmenting an existing formal specification language to make it usable in statistical testing will be a step towards this reduction. This language extension has two main objectives, (1) integrate the theory and application of statistical testing within a formal methods software engineering framework, and (2) evaluate the relative efficiency of uniform statistical testing and structural testing. By improving usage-model building, we hypothesize that we can test systems of significantly larger size and complexity than current methods for statistical testing. We test our hypothesis and accomplish the objectives of this research by pursuing the following specific aims which are the contributions of this dissertation:

1. **Use hierarchical and parallel state models for automated statistical test case generation.** Our working hypothesis is that it is possible to implement parallel model statistical testing by extending an existing formal modeling notation, thereby providing an additional valuable tool in a well-established software engineering framework. This requires us to develop techniques to identify and assign appropriate probabilities to RSML^{-e} models.

2. **Test the efficiency of statistical testing and structural testing.** Several hypotheses are tested for this aim, to include the following:

Hypothesis 1: Given an equivalent number of input messages, uniform statistical testing using many short test cases provides greater state, transition, and transition decision coverage than uniform statistical testing using fewer long test cases.

Hypothesis 2: Given an equivalent number of input messages, uniform statistical testing using many short test cases finds more faults than uniform statistical testing using fewer long test cases.

Hypothesis 3: Given equal effort in generating and applying test cases, uniform statistical testing can provide equivalent levels of state, transition, and transition decision coverage, compared to structural coverage testing designed to provide transition decision coverage.

Hypothesis 4: Given equal effort in generating and applying test cases, uniform statistical testing can detect equivalent numbers and types of faults, compared to structural coverage testing designed to provide transition decision coverage.

1.1 Contributions

This research is innovative, as it provides a new method of modeling and testing increasingly complex systems. To our knowledge, there is currently no methodology that provides statistical testing based on parallel models. To achieve this, we have developed techniques for attaching probabilities to parallel models that are currently nonexistent in the statistical testing arena. Furthermore, we create a testing framework that enables testing several of the claims made concerning the faultfinding capabilities of statistical testing as compared to other testing strategies.

The results are significant because they will contribute to reducing the high costs

of testing, while not requiring a high level of expertise in setting up the testing. Secondly, they will provide the capability to drive the statistical testing with parallel models, which will allow us to build usage models significantly larger than what was previously possible, since parallelism allows more compact expression of system behavior.

Finally, our results advance the knowledge of modeling and statistical testing in software engineering. We started with a software engineering environment that supports development, analysis, and execution of formal requirements models, which presented an excellent starting point for integrating statistical testing applications. We added statistical testing components and now have a software engineering environment that allows further exploration of statistical testing using more complex models.

1.2 Organization of the Dissertation

This dissertation is comprised of two distinct parts, the development of a statistical testing framework for parallel models in Chapters 2 through 4 and the evaluation of that framework in Chapters 5 through 6.

The statistical testing framework development portion is arranged as follows, Chapter 2 addresses related work in statistical testing and state-based specification languages, highlighting the contributions they have made to software engineering and the limitations that we had to overcome with this research. Chapter 3 lays out our extensions to a requirements specification language that allow statistical testing of larger, more complex systems. We define what the probabilities mean in a real world system and illustrate operational profile development through a running example. Chapter 4 completes this section by outlining the design of the testing framework and how we constructed it.

The evaluation of this statistical testing framework for parallel models is presented in Chapters 5 and 6. Chapter 5 presents an overview of the evaluation of this research effort and the questions we set out to answer, showing that this testing framework is possible and evaluating its efficiency relative to structural testing. Chapter 6 provides the results of our research.

Finally, we summarize the research of this dissertation and present the future direction of our research efforts in Chapter 7.

Chapter 2

Related Work - Languages

Several areas of related work are fundamental to this research. In this section, we outline the problem with effective, efficient testing, and then describe statistical testing and state-based specification modeling languages and how they can be used to lessen the testing burden.

As noted in the previous chapter, testing is a very difficult and time-consuming task in software development. It is not possible to test all combinations of inputs or behaviors of a software system since the range of input may be enormous and there are essentially infinite combinations of inputs that must be investigated [5]. To address the testing problem several different strategies have been suggested. One popular strategy is to attempt to partition the input domain into subdomains based on the requirement on the software—domains in which all inputs from a particular domain will exercise the same part of the software system under test [36]. By picking one representative input from each domain, we will exercise all parts of the software. This partitioning methods suffers from the need to have expert knowledge of the domain to make the partitioning. Also, performing a perfect partitioning is not possible in general and there will be behaviors of the software that will never get tested [69].

A complementary approach is based on finding test cases based on the structure of the software under test. The goal is to generate tests that provide structural coverage of the code up to some coverage criterion, for example, all statements, all branches, all conditions made true and false, etc [28, 50]. Again, there are many drawbacks with

this testing approach. Most seriously, it is based on the implementation and, thus, decoupled from what the software is really intended to do, that is, the requirements. In addition, covering various structures of the software does not guarantee that any faults will be revealed. Finally, both approaches discussed above devote the same amount of effort to all parts of the problem domain or the software—the same amount of effort is devoted to a portion of the software regardless of how much it will be used during operation. If a section of the system is rarely used, it will get as much testing as a section that is used extensively. Given the difficulty of testing, we will not remove all faults from the software; thus, we will leave faults in all parts of the software regardless of how much it is used. If we instead focused more testing effort on the often used portions of the software, we will remove more faults from this part and the user will be less likely to encounter any faults—by redirecting the effort, we may leave the same total number of faults in the software, but, since the portions most used have been tested the most, the user will perceive the software to have better quality than software that has been subjected to uniform testing. In this statistical approach to testing, an operational profile (or usage profile) is developed to guide testing to the portions of the system that need the most testing. As the number of test cases grows, the little used sections get tested, just not to the level of the most-used sections.

In this chapter, we provide an overview of statistical testing, outlining how Markov chain models can be used for testing and analyzing software systems. We also review existing Markov chain model tools, presenting an example of a tool and language used for statistical testing. This example is illustrative of the benefits of statistical software testing but also gives evidence of its limitations.

Lastly in this chapter, we discuss state-based specification languages, which provide the possibility of significant reduction in the size of a system model through the use of parallelism. This capability can be used to mitigate the model size and

complexity issue for statistical testing.

We merge these concepts in our process to leverage the strengths of all of them, producing a capability to create significantly more complex operational profiles and statistically generate test suites from them.

2.1 Statistical Testing

Statistical testing is based on the use of an operational profile for generating test cases and analyzing a system's behavior [56, 57, 72, 73]. Statistical testing executes a system under test according to this operational profile to test the system as it would be used in its operational environment.

The operational profile indicates the probability of a specific input arriving to the software given the sequence of previous inputs that have already arrived and the behavior of the system based on that input. The operational profile is composed of two components, structural and statistical, as shown in Figure 2.1. The structural component describes the modeled system's possible behavior and the statistical component describes what the probability of given inputs from the environment are expected to be given the state of the system.

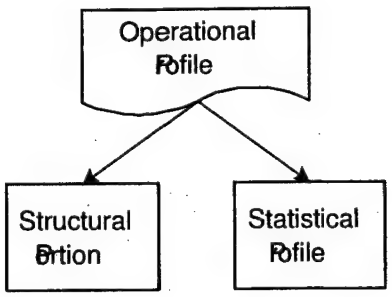

Figure 2.1: Operational Profile Components.

The operational profile of the system under test can be modeled as a Markov chain,

a state-based model, with each transition out of a state labeled with the probability of the associated exit stimulus occurring from that state. This Markov chain is often refered to as a usage chain, as it details the intended usage of the system. Intuitively, one can see that the sum of all transitions out of a state must equal one.

Once the Markov chain model is created and probabilities are assigned to all of the transitions, the model can be subjected to static and dynamic analysis, as shown in Figure 2.2. Static analysis, covered later in this section, produces metrics such as the expected appearance rate of a given state in the long run, the number of states generated until first occurrence of a given state, and expected test case length (number of states generated) [56, 73]. Dynamic analysis, which involves executing the model, creates a data structure similar to the usage chain, which is called the test chain. The test chain incorporates the states of the usage chain but also includes new states and arcs created by failures in the system. The actual usage of the software is recorded in the testing chain and then the test chain is compared to the usage chain. As more tests are conducted, the testing chain begins to look more like the usage chain. Faults cause more deviation of the testing chain from the usage chain; successes result in more similarity. The amount of divergence between the two chains, measured as the log probability of the two Markov chains, can signal the end of testing, when the divergence falls below a acceptable level. Reliability can also be measured from the testing chain by making the termination state an absorbing state and calculating the probability of entering the termination state. Other metrics available from dynamic testing include test sufficiency, reliability, expected number of steps between failures, and entropy, the uncertainty present in the model [27, 31, 58, 67, 74].

Benefits There are several benefits to using usage model-based statistical testing [27], as follows:

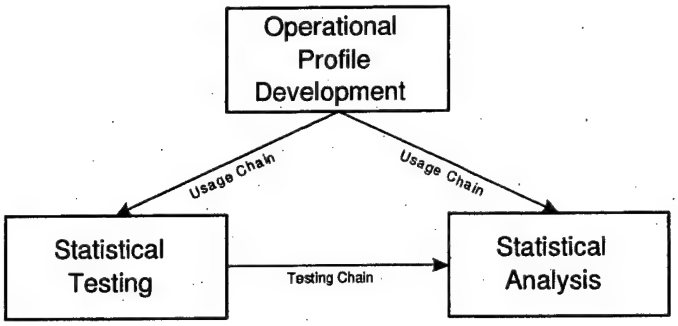

Figure 2.2: Operational Profile Development Process.

- Validation of requirements - models provide an easy to understand picture of the specification, allowing all interested parties an understanding of the process and leading to early discovery of errors or omissions.

- Resource and schedule estimation - metrics such as number of sequences to cover all arcs/states can lead to a refined testing schedule to make the most of whatever time is available.

- Automated test generation - as noted earlier, statistical testing is ideal for automated test generation. The automation can even create various test strategies, such as all arcs/states/paths coverage or straight statistical tests, based on the Markov chain information.

- Effective, efficient testing - minimize the testing effort while ensuring all testing requirements are met; put effort where it will do the most good; metrics also provide a method to determine when to stop testing, besides the typical deadline approach

- Quantitative test management - solid data is available to make business management decisions.

The above list of benefits shows that statistical testing is a promising method for determining software quality. However, as will be detailed later, there are several areas for improvement in the methodology.

2.1.1 Markov Chain Modeling

The culmination of the statistical testing research effort thus far has led to a basic framework for producing and using Markov chain models in statistical testing [27, 47, 46, 58, 63, 72, 73, 74]. The primary steps in the methodology include analyze the system, build the model, determine probabilities, generate test cases, and run analyses on the model. These are further defined in the next few sections, followed by the limitations of the statistical testing methodology.

Analyze System: The first step in any development problem should be analyzing the system to be developed, to identify how the system will be used. This includes the domains of interest, the procedures already in place (if applicable), and any modifications to be made. This information can be found through inspection, specification, or both. By interviewing experts of the domain or following the processing in the domain, it is possible to determine much of the structure and behavior of a system. In the case of many software applications, such as graphical user interfaces, the externally visible actions are what should be modeled in the Markov chain model [31, 44, 56]. These actions can be discovered and recorded by examination of an existing system or by observing experts at work.

It is also possible that the behavior of the system, existing or planned, may already be documented in a specification or requirement document. This can also be a good starting point for development of a model; however, it is fairly certain that, even with a substantial document, interaction with experts will be necessary to finalize

the resultant model. The specification of a system, such as created with a formal specification language such as RSML^{-e}, already delineates most of the information needed to build a Markov chain model. As noted earlier, we developed a process for extending a RSML^{-e} specification of a system to become an operational profile. This saves significant effort in operational profile modeling, as the formal specification model already exists and is used for several other software engineering tasks.

Build Model: The actual construction of the model is the crux of the issue, yet it remains essentially a creative process [73]. The earliest statistical testing papers [1, 27, 73] refer to the construction of two models of the system, the usage chain and the testing chain. The usage chain is a representation of how the system is expected to behave in normal operations and the testing chain is the recording of how the system actually performs under test. These two chains are the basis for most of the analysis, yet the construction of them is more of an art than a science. The models are typically presented as directed graphs, with the nodes representing states and the arcs representing transitions between states. An example of a Markov chain model is found at Figure 2.3, with states identified as numbered circles and transitions as arrows between states.

In this model, the system starts in State 0, and transitions to State 1 upon receiving the input "Init." From State 1, the system can transition to State 2 or State 3, depending on the input received, "X" or "End," respectively. These two inputs occur with equal probability, shown by the 0.5 value on both transitions. Finally, State 3 is a terminating state, as it has a transition in but no transition out.

Markov chain model construction is a hierarchical process [73]. A software model that exists at the highest level consists of three states, invocation, usage, and termination. From this small model, the usage state can be broken down into many new, smaller, less abstract states until the desired level of detail is obtained. The input

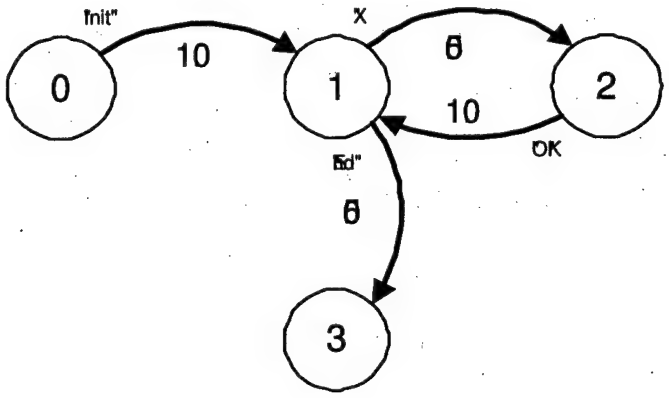

Figure 2.3: Example Markov Chain Model.

stimulus occurrence probabilities are then added as described above.

A testing chain is the record of the behavior of the software system under test. It starts with a copy of the structural portion of the usage chain and includes failure information by adding failure states with the appropriate transitions connecting them. Two exits from the failure state are possible, 1) a complete failure, which transitions to a termination state, or 2) a partial failure, which transitions to some other, valid state. As the testing progresses, the transition occurrence counts are updated as they become active and the probabilities are calculated at the end of testing. The resultant testing chain is thus a copy of the usage chain with included failure states and updated arc transition occurrences [73].

Determine Probabilities Determining the probabilities to assign to each arc of the model is one of the broadest areas of study in this methodology. The source of probabilities is of greatest concern here. Three possible sources of probabilities are labeled uninformed, informed, and intended [73]. The *uninformed* approach simply labels the arcs out of a node with a uniform distribution. It is so named as it represents a lack of better information about the arc probabilities. This is a good place to start,

when field data is not known, but results in the highest entropy. The *informed* approach uses actual user data, which can result in different models, depending on whether the model represents a single user, a small class of users, or a compilation of all users. This approach gives the most accurate results as it uses actual field data. The actual user data can be gathered by monitoring existing systems or by consulting domain experts. Automating the collection by instrumenting the existing code and recording actual transitions would be the most accurate method for collecting such data, but no evidence is found as to this being done or how to do it. The limitation of interviewing users is the possibility of skewing the data due to user bias—what they think they do the most is not necessarily what they actually do the most. Finally, the *intended* approach can produce multiple models, based on who or what is being represented as the user, but it shows the expected usage of the system under test by "a careful and reasonable user" [73] rather than the actual usage by users in the field.

In addition to explicitly assigning probabilities, two types of constraints can be used to assign probabilities to Markov chain models [45, 46]—relational constraints, in which there is some relationship between two constraints or probabilities, and objective functions. The first allows for more flexibility in the values of transition probabilities but retains the relationship between them. The Model Language (TML) [46] was designed to accept all three types of probability assignment in a Markov chain model and can incorporate constraints in multiple mathematical programming languages. The Java Usage Model Builder Library (JUMBL) [38] was also designed and created by the SQRL and can analyze Markov chain models with all of these assignment types. TML and JUMBL are further described in Sections 2.1.2 and 2.1.2.

With the explicit probabilities and the constraints, the system model is complete as an operational profile. The most common representation method of an operational profile for statistical testing is the transition matrix [9, 45, 73]. By constructing an

n x n matrix, where n is the number of states, each transition can be represented, as in Table 2.1. These data are the probabilities from Figure 2.3. $P_{i,j}$ is the probability of a transition from state i to state j. The lack of a transition between two states (including a state and itself) is given a 0 ($P_{i,j} = 0$) and a mandatory transition is given a 1 ($P_{i,j} = 1$). All other probabilities are determined as noted in the previous paragraph and are entered appropriately in the matrix. The major drawback in the transition matrix method is the space requirements for the table, especially when the matrix is sparsely populated. For strongly connected graphs this is an efficient method, but for sparse graphs it wastes space with zero probabilities. Once the matrix is complete, it is usable for calculating statistical testing metrics.

Table 2.1: Transition Probabilities Matrix.

$P_{i,j}$	0	1	2	3
0	0	1	0	0
1	0	0	.5	.5
2	0	1	0	0
3	0	0	0	1

The matrix can be written up in a modeling language for statistical testing and analysis, such as TML. TML is the language we will use in some examples later in this dissertation.

Generate Test Cases: Markov chain modeling is well suited for use in generating test cases. The general idea is to use the transition matrix (or other graph representation) to choose the exit arc (transition) from the current node (state). A random number generator is used to determine which exit arc is chosen from each state. From these traversals of the matrix or model, a set of test cases can be generated that maximizes testing of the portions of code most used by consumers.

Analyze Model: Static and dynamic analysis can be performed, either on the model itself or on the results of running test cases. Static analysis metrics [47, 73], which are performed on the model itself, include expected appearance rate of state I in the long run, number of states generated until first occurrence of state I, expected test case length, number of test cases until state I occurs, entropy (uncertainty present in the model), expected percentage of states covered at sequence n, expected percentage of arcs covered at sequence n. These metrics are particularly good for management activities, to plan how best to use an organization's resources on a project.

Dynamic analysis of testing [27, 31, 58, 67, 74] is possible for the following metrics, test sufficiency, reliability, expected number of steps between failures, absence of failures in a test run, approximate equality measure, and entropy. These metrics are for determining the quality of the software being tested and for determining when to stop product testing.

These steps boil down to three distinct activities in statistical testing, (1) building models, (2) generating tests, and (3) analyzing models. For each of these activities, there are problems that merit further study, such as

1. Building models is limited by the complexity of the systems to be modeled. As new components are added, greater than linear increases in the size of the model are often encountered, rendering the modeling effort for statistical testing difficult or intractable. Also, the use of infinite domains, such as integers and reals, can cause models to become intractable.

2. Generating tests is limited primarily by the modeling issue. If the model is too large to fit in the generating computer's memory, it is unlikely we can generate valid tests from it.

3. Analyzing models becomes intractable very quickly, as the size of the matrix

required for analysis grows exponentially with added states. This quickly overwhelms the analysis application.

This dissertation addresses the model building and test generating activities of statistical testing. In the following sections, we review several statistical testing tools and languages, highlighting the strengths and limitations of each and, in turn, of statistical testing in general. In the final section of this chapter, we describe state-based specification languages, which we use to reduce the modeling burden.

2.1.2 Markov Chain Model Tools

There are many Markov chain modeling tools available, but one prominent tool is the Java Usage Model Builder Library (JUMBL), with its accompanying model definition language, The Modeling Language (TML) [46]. We present them here as representative of these tools and languages, then provide highlights of other tools.

Java Usage Model Builder Library (JUMBL)

The following information is adopted from [38] and describes how to use the Java Usage Model Builder Library (JUMBL). The utilities are discussed in the order they would typically be used. More extensive documentation is available in the user's guide which ships with the JUMBL.

Creating Models: You may develop your model in any file format understood by the JUMBL. The suggested format is The Model Language (TML). TML is a textual format designed specifically for representing Markov chain usage models, and supports hierarchical modeling, constraints, and automated testing information.

Checking Models: JUMBL has a command `Check` that checks the structure of your TML models. Check verifies that there is a path (a sequence of arcs) from the model source to every node of the model. If a node is not reachable from the source, it is listed as "unreachable." Check then verifies that there is a path from every node of the model to the sink. If the sink cannot be reached from a node, the node is listed as a "trap" node. Check also reports the number of nodes and arcs in the model and verifies input trajectories and selectors.

Dealing with Hierarchical, Referenced Models: If you have built a hierarchical model out of component models ("referenced models"), you may wish to "flatten" the model (reduce it to just states and arcs, with all model references either removed or instantiated). Valid models must exist for all referenced models.

Analyzing Models: The `Analyze` command produces an HTML report that describes the statistical properties of the Markov chain. Analyze resolves constraints on the model, if any are present. By default the unnamed distribution is used. You can specify a different probability distribution by specifying the key. The distribution key must be defined in the model.

Generating Tests: Tests are generated by the `GenTest` command.

- `GenTest` by default generates random walks of the model, conditioned by the specified probabilities (if any).
- `GenTest --min` generates a non-random sample of tests which cover all arcs of the model with the lowest cost (if no cost is specified, 1.0 is assumed for every arc and thus the model is covered in the fewest steps possible).

- `GenTest --weight` generates non-random tests in decreasing order by probability mass, with the highest first.

You can specify the number of tests for GenTest, except when using the `--min` option.

When using GenTest, you may want to select one of many probability distributions. By default, the unnamed distribution is used. You can select another by specifying the appropriate key.

Recording Results: Test results are recorded in the test cases themselves. By default, this records that all events in every test were completed successfully. Test failures are recorded by step number and whether it resulted in the test stopping.

Test Analysis: Once you have recorded the results of the testing experiment, you can use `AnalyzeTest` to get a detailed analysis of the testing experience, including expected field reliability. This produces an HTML report containing information about the testing experience. By default, the unnamed ("default") distribution is used. If you want to specify a different probability distribution, you can specify the key.

The Model Language (TML)

TML is a prime example of a modeling language designed for describing statistical testing Markov chains. The information presented here is derived from [46].

The Model Language (TML) was developed for describing usage models, also known as operational profiles, for statistical testing. Usage models are designed as state models, with arcs between states triggered by events. In this section, we will describe the basics of the TML language, with accompanying examples.

Models: Models in TML are introduced by the `model` keyword, as shown below. A model name must follow the `model` keyword, and may consist of upper- and lower-case letters, digits, and the underscore.

```
model modelname
```

The general form for a model definition is:

⟨*model definition*⟩ ::=
'`model`' ⟨*model name*⟩
(⟨*state definition*⟩) *
'`end`'

States: The general form for each state definition in a model is:

⟨*state definition*⟩ ::=
'['⟨*state name*⟩']'
(⟨*arc definition*⟩) *

Every model has two special states: a *source* and a *sink*. The default values for these states are `[Enter]` and `[Exit]`, respectively. The source state is the initial state of the model and the sink is the terminating state of the model. The sink state cannot have any outgoing arcs and therefore, TML does not require explicitly stating `[Exit]`. Other state names may be substituted for the source and sink by prefacing them with the `source` or `sink` keyword.

Arcs: The general form for an arc definition is:

⟨*arc definition*⟩ ::=
'"'⟨*event name*⟩'"' '['⟨*state name*⟩']'

Events: Events represent some action in the software under test. Each sequence of events determines a unique path through the model and thus has an easily determined probability of occurrence. All outgoing arcs from a given state must have distinct event names and all state names for a given model must be unique. This restriction also has the advantage of helping the user detect structural problems.

With these structures, we can create the TML model for the Markov chain model in Figure 2.3, shown in Figure 2.4.

```
model A
source [state0]
  "Event1"      [state1]
[state1]
  "Event2"      [state2]
  "Event3"      [state3]
[state2]
  "Event4"      [state1]
sink [state3]
end
```

Figure 2.4: Example TML Markov Chain Model.

Model Composition: Models can be combined hierarchically, much like using subroutines in programming languages. In this manner, levels of abstraction can be obtained by including or excluding those subroutines. In the arc definition, the name of the model to call like a subroutine is placed after the event and before the destination state. The general form of this alternate arc definition is:

⟨*arc definition*⟩ ::=
‘ “ ’ ⟨*event name*⟩ ‘ ” ’ ⟨*model reference*⟩ ‘ [’ ⟨*state name*⟩ ‘] ’

In this case, the event triggers a jump to the submodel and, upon exiting the submodel, the destination state is entered. As an example, `event1` causes a transition

from `state1` to `state2` with `model2` included, and is constructed as follows:

```
[state1]
    "event1" model2 [state2]
```

Constraints: The preceding constructs of TML are all structural. Usage models also require statistical constructs. Constraints can be placed on the arc, state, or model definitions to provide the necessary probability distributions for analysis of the model. It is possible to use any of several constraint languages for describing these constraints. Whatever language is used, constraints will always be found in TML in a ($...$) structure. The scope of a constraint, whether it applies to the model, a state, or an arc, is determined by what the constraint precedes. A constraint precedes the name of the item it constrains. There are three types of constraints possible, shown below:

1. **Fixed value** - This is the simplest constraint, where a fixed probability value, such as ($ 0.60 $) or ($ 7 $), is attached to the event. If one event is five times more likely to occur than another event, they can be given constraints ($ 5 $) and ($ 1 $), respectively. The TML model is usually given a `normalize` constraint to normalize the constraints to some given value, usually `($ normalize (1) $)` (for probabilities) or `($ normalize (100) $)` (for percentages).

2. **Variables** - Constraints can use variables to help define the relationships between constraints. One potential use for this construct is to define a variable at the model level, making it a global variable, then using the variable on events in the states. In this manner, the variables could be changed once, but affect the entire model, rather than having to go in an manipulate each arc definition individually.

3. **Arithmetic** - It is also possible to use arithmetic formulas within constraints to define their values. Examples are as follows:

   - (`$ a=0.1 $`) // A constant, a = 0.1
   - (`$ b=a*2 $`) // Another constant, b = 0.2
   - (`$ 1/a $`) // Constraint = 10
   - (`$ a^2*100 $`) // Constraint = 1
   - (`$ b-a*2 $`) // Constraint = 0

The following functions select values when an event is unconstrained.

- `assume()` - whatever value is assigned between the parentheses is used for any unconstrained event. Commonly, `assume(1)` is used.
- `fill()` - whatever value is found between the parentheses must be the total of all exit arcs from any given state. Constrained arcs will be subtracted from the given value and the remaining value is distributed uniformly across all remaining, unconstrained arcs. Here, `fill(1)` or `fill(100)` can be used for probabilities and percentages, respectively, if some values are known and the others are not of great concern.
- `emax` - The value of unconstrained arcs will be set so as to maximize the entropy of the probability distribution. For more information on entropy, see [67, 73, 74].

Keys: Models may be designed with more than one probability distribution, to aid in differeent forms of analysis. For example, there may be different usage patterns for different users or developers may wish to test for security purposes, anticipating malicious users. In this case, a *key* is used to identify constraints as belonging to

particular distributions. This is accomplished by adding a key word, consisting of upper- and lower-case letters, numbers, and the underscore, followed by a colon, in front of the constraint:

```
kEy_WoRd1: ($ assume(1) normalize(1) $)
KeY_wOrD2: ($ emax normalize(100) $)
model example_model
      :
end
```

This gives us two distributions, the first normalizes to 1 using 1 as the fill in value for unconstrained arcs. The second normalizes to 100, but fills in values for unconstrained arcs to maximize the entropy of the model.

Now we can show that the general form of a constraint is:

⟨*constraint declaration*⟩ ::=
    (⟨*key*⟩ ':') '($' ⟨*constraint*⟩ '$)'

These are the primary constructs of TML. For our purposes, these constructs will be sufficient to construct usage models from RSML^{-e} specifications. Our simple model from Figure 2.3 is completed with all these components in Figure 2.5. For further information on these constructs or for additional constructs, the reader is encouraged to consult [46].

TML and JUMBL are valuable tools for software testing but are limited by the kinds of systems they can model. In our experience, it is very difficult to model any system of realistic complexity. Our research overcomes this limitation by using parallel constructs in a state based model.

TML with JUMBL is one method to implement statistical testing. However, it suffers from a few shortcomings. The first is the size and complexity of models that can be used. TML can model flat and hierarchical systems, but even the hierarchical models must be reduced to flat models before they can be analyzed by JUMBL and

```
($ normalize(1) $)
model example
source [state0]
  ($ 1.0 $) "Init" [state1]
[state1]
  ($ 0.5 $) "X"    [state2]
  ($ 0.5 $) "End"  [state3]
[state2]
  ($ 1.0 $) "OK"   [state1]
sink [state3] end
```

Figure 2.5: Complete Example TML Markov Chain Model.

test cases generated. A system of any reasonable size or complexity simply cannot be modeled without overwhelming the test case generation and analysis machinery. Our use of a state-based specification model allows modeling and testing of much larger, more complex systems. Below we briefly discuss additional tools somewhat related to the research discussed in this dissertation.

Symbolic Hierarchical Automated Reliability and Performance Evaluator (SHARPE): [24] - SHARPE assesses performance, reliability, and availability of Markov chain systems. It has syntax similar to that of TML and performs analysis at the system level, where the behavior (reliability) of the individual components is already known. We are interested in a lower level of abstraction, determining the reliability of the individual components as well as the system itself.

Markov Chain Analyzer (MARCA): [60] - MARCA is used primarily in the telecommunications arena, but also for reliability modeling of hardware and software systems. However, it is designed for modeling at the system level, whereas we are interested at the behavior of the internals of the components. MARCA does claim the ability to store the transition matrix compactly, which is of interest for later phases

of our long-range research. It works on flattened models, so it will suffer from the same complexity issues in the analysis portion of statistical software testing.

Erlangen-Twente Markov Chain Checker (ETMCC or ETMC2): [26] - ETMCC performs symbolic model checking on Markov chain models. It uses the temporal languages Probabilistic Computation Tree Logic (PCTL) and Continuous Stochastic Logic (CSL). We are not performing symbolic model checking in our research, so its analysis engine is not of interest to us. However, it also has a numerical engine that solves linear systems, which may be of use to us in the analysis portion of our long-range research.

Performance Evaluation of Parallel Systems (PEPS): [12] - PEPS is an application for evaluating parallel and distributed communications systems through the solution of Markov chains. As in SHARPE, MARCA, and ETMCC, PEPS is primarily concerned with higher levels of abstraction than those we are interested in. One assumption of PEPS that is not necessarily valid in our modeling is the idea that the parallel system is made up of many independent components that communicate only rarely with each other. This is often true in the telecommunications arena that PEPS was designed for but does not match the reality of the systems we will model. Software modeled with parallel components, like those specified in RSML^{-e}, may be significantly interconnected. The minimal communication between components is what allows PEPS to model the system as individual Markov chain models. This drastically reduces the amount of space the transition matrices require for storage. Communication between modules is modeled as two additional matrices for every two-way communication. Using this paradigm with our software models would reduce the storage required for the transition matrices but significantly increase the amount of extra storage required for the communications matrices. PEPS does, however, use

tensor algebra [12] to reduce the size of the transition matrices and simplify the computations, which may be of use to later research.

None of the above Markov chain packages is directly usable in our research. However, many of the concepts they employ, such as the tensor algebra in PEPS and the numerical engines of MARCA and ETMCC, may be adaptable to later phases of our research.

Statistical testing promises significant improvements in testing efficiency but suffers from some drawbacks. Markov chain models are required to be flat state-based models, which grow to immense size as systems get larger and more complex. Also, for models that have small probabilities on inputs, it may take an incredibly large number of test cases to get coverage of some portions of the code. In these cases, alternate input distributions can be used to ensure test case coverage of low-probability, high impact sections of systems, such as nuclear power plant fail-safe systems (you hope never to use it, but if you do, it had better work).

2.2 State-based Specification Languages

As the size of the model is key to performing statistical testing on models of realistic complexity, overcoming the model size problem is a primary impediment to improvements in statistical testing. Markov chain modeling relies on flat models, which can grow exponentially with added states. We believe formal modeling languages designed to deal with large state spaces hold the key to mitigating the statistical testing model size problem. There are many specification languages available, including SCR [22], Z [59], STATEMATE [18], and RSML^{-e} [19]. In this section, we review RSML^{-e} as representative of these state-based specification languages.

2.2.1 RSML^{-e}

The following description of RSML^{-e} and its component parts is adopted in large part from [71] and [65]. Examples of the components can be seen in the model in Figure 2.6 and the code in Figure 2.7.

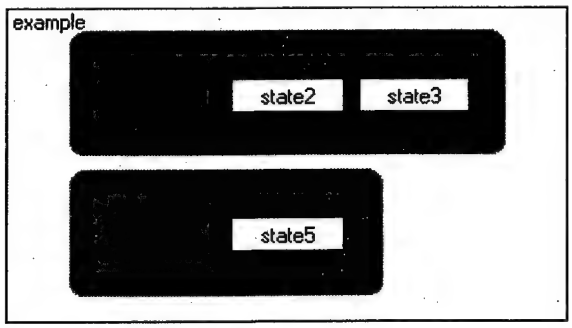

Figure 2.6: Example RSML^{-e} Model.

RSML^{-e} enables simulation, code generation, and visualization of a software specification, using the NIMBUS environment [64], a flexible, extendible environment for specification of safety-critical systems. RSML^{-e} and NIMBUS provide a language and environment for software developers to specify safety critical systems with high reliability. RSML^{-e} is readable, which aids in manual inspections and reviews; it is also suitable for formal analysis and simulation, using the requirements model in the NIMBUS environment.

RSML^{-e} is based on the language Requirements State Machine Language (RSML) developed by the Irvine Safety Research Group under the leadership of Nancy Leveson [34]. RSML^{-e} was developed as a requirements specification language specifically for embedded control systems. One of the main design goals of RSML^{-e} was readability and understandability by non-computer professionals such as users, engineers in the application domain, managers, and representatives from regulatory agencies [34]. RSML^{-e} is based on hierarchical finite state machines and dataflow languages. Visu-

```
STATE_VARIABLE A :
   VALUES : {state1, state2, state3}
   PARENT : None
   INITIAL_VALUE : state1
   CLASSIFICATION: State
     Transition state1 TO state2 IF
       TABLE
         Var1    : T ;
         Var2    : F ;
       END TABLE
     Transition state1 TO state3 IF
       TABLE
         Var1    : T ;
         Var2    : T ;
       END TABLE
   EQUALS state1 IF not(Var1)
END STATE_VARIABLE
STATE_VARIABLE B :
   VALUES : {state4, state5}
   PARENT : None
   INITIAL_VALUE : state4
   CLASSIFICATION: State
     Transition state4 TO state5 IF Var3
     Transition state5 TO state4 IF not(Var3)
END STATE_VARIABLE
IN_VARIABLE Var1 : Boolean
  INITIAL_VALUE : FALSE
  CLASSIFICATION : MONITORED
END IN_VARIABLE
```

Note: Var2 and Var3 are defined like Var1.

Figure 2.7: Example RSML^{-e} Model Code.

ally it is somewhat similar to David Harel's Statecharts [17]. For example, RSML^{-e} supports parallelism, hierarchies, and guarded transitions. The main differences between RSML^{-e} and RSML are the addition in RSML^{-e} of rigorous specifications of the interfaces between the environment and the control software, and the removal of internal broadcast events.

An RSML^{-e} specification consists of a collection of input variables, state variables, input interfaces, output interfaces, functions, macros, and constants, which will be briefly discussed below.

In RSML^{-e} the state of the model is the set of assignment histories of all variables and interfaces. The state information is used to compute the values of a set of state variables, similar to modes in SCR [23]. These state variables can be organized in parallel or hierarchically to describe the current visible state of the system. Parallel state variables are used to represent the inherently parallel or concurrent concepts in the system being modeled. Hierarchical relationships allow child state variables to present an elaboration of a particular parent state value. Hierarchical state variables allow a specification designer to work at multiple levels of abstractions, and make models simpler to understand.

Assignment relations in RSML^{-e} determine the value of state variables. These relations can be organized as transitions or condition tables. Condition tables describe under what condition a state variable assumes each of its possible values. Transitions describe the condition under which a state variable is to change value. A transition consists of a source value, a destination value, and a guarding condition. A transition is taken (causing a state variable to change value) when (1) the state variable value is equal to the source value, and (2) the guarding condition evaluates to true. The two relation types are logically equivalent; mechanized procedures exist to ensure that both functions are complete and consistent [21].

Conditions are simply predicate logic statements over the various states and variables in the specification. The conditions are expressed in disjunctive normal form using a notation called AND/OR tables [34]. The far-left column of the AND/OR table lists the logical phrases. Each of the other columns is a conjunction of those phrases and contains the logical values of the expressions. If one of the columns is true, then the table evaluates to true. A column evaluates to true if all of its elements match the truth values of the associated columns. An asterisk denotes "don't care."

Input variables in the specification allow the analyst to record the values reported by the environment or various external sensors. They are assigned based on the messages received by input interfaces.

Interfaces encapsulate the boundaries between the RSML^{-e} model and the external world. There should be a clear distinction between the inputs to a component, the outputs from a component, and the internal state of the component. Every data item entering and leaving a component is defined by the input and output variables (state variables designated as outputs). The state machine can use both input and output variables when defining the transitions between the states in the state machine. However, the input variables represent direct input to the component and can only be set when receiving the information from the environment. The output variables are presented to the environment through output interfaces.

The state variables are placed into a partial order based on data dependencies and the hierarchical structure of the state machine. State variables are data-dependent on any other specification entities that are contained in the predicates in their condition tables. A variable is also data dependent on its parent variable (if it has one). The value of the state variable can be computed after the items on which it is data dependent have been computed. A single computation of all the variables in the specification is referred to as a step.

A RSML^{-e} step occurs whenever R (the next state relation) is recomputed [71]. A step begins because of a change in the external environment-either the receipt of a new message by an input interface, or an update from the system clock. The behavior of the next-state relation can be seen in four stages.

1. The external environment changes, causing a message to be received on an input interface, and/or causing the system clock to be updated.

2. The input interfaces decide which input variables (if any) to update based on the change in the external environment.

3. The values of all state variables in the machine are recomputed, yielding the current state of the machine.

4. New output messages may be generated and sent to the external environment by the output interfaces.

Our research is concerned with the first three of these stages, since they determine the next state of the machine. The next state relation is created by composing the assignment relations. The order in which the relations are composed is determined by the data dependencies of each StateObject in the specification. At a high level, the data dependencies of an object y describe what must be computed before the value of y can be computed. The data dependencies for an input variable are the set of input interfaces that can assign the variable a value. The data dependencies for an input interface, state variable, or output interface are the other variables or interfaces referenced in the assignment relation, with some exceptions. An example of a data dependency graph for Figure 2.6 and Figure 2.7 can be found in Figure 2.8.

The RSML^{-e} code throughout the rest of this paper will refer to the models in Figure 2.6 and Figure 2.7. Note that the code examples shown in this paper do not show the interfaces, so as to keep the examples concise and clear.

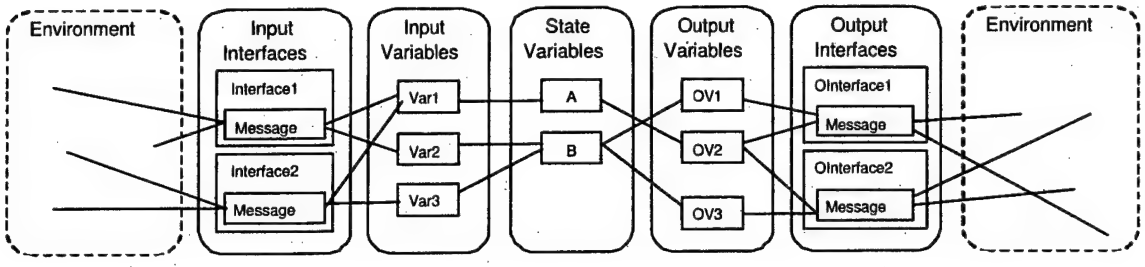

Figure 2.8: Example Assignment Dependency Graph.

2.2.2 Benefits

Specification languages like RSML^{-e} not only help to formalize all of the customers requirements, but also give the ability to perform simulation, code generation, model checking, theorem proving, and structural test case generation. A development environment like NIMBUS could benefit greatly from the integration of an additional tool like statistical testing, which would improve our testing capabilities. In this dissertation, we bring the knowledge of the world of specification languages to merge with the knowledge of statistical testing and produce a new capability—statistical testing of parallel models.

Chapter 3

Language Extension

To extend the capabilities of statistical software testing, without placing a huge burden on the software developer, we chose to merge the knowledge of statistical testing and specification languages. In this chapter, we explain the concept behind transforming or extending existing software artifacts and the extensions to RSML^{-e} that we developed to allow for modeling significantly larger systems than is currently possible by other statistical testing tools.

3.1 Transformation vs. Extension

It is evident that many of the components of an operational profile are found in a requirements specification as well. For example, both describe the behavior of the system and thus, both have transitions defined. Since there is much in common, it should save resources if a method for reuse or transformation of existing software artifacts could be found or developed. Regnell, *et al.* [54], describes using use-case modeling to develop both a requirements specification and an operational profile for statistical testing. The results of their research are two methods of action, transformation of a use-case model to an operation profile or extension of the use-case model to *be* the operational profile. The benefits of these choices are a reduction in the modeling effort and added traceability from requirements to testing, which should reduce maintenance costs.

The transformation method would convert the use-case model to an operational format, such as TML, which would allow the use of analysis machinery that already exists. The extension method results in a single model being used for two purposes, use-case and operational profile, thus requiring less modeling effort and potentially less maintenance expense. However, if the difference between the use-case model and the operational profile is large, more compromise may be required, resulting in a larger model with more extraneous information for either purpose.

Our research is along similar lines, except we work with a formal specification rather than a use-case model. Since we already have a requirements specification environment and process established, we leverage what exists by adding statistical testing to it. We, too, are faced with two possibilities, to transform our requirements model to an operational profile in an existing statistical testing language, like TML, or extend our specification language and environment to allow us to use our model as the operational profile and perform the statistical testing directly on the model, within our own environment rather than with an external application. The former method, transformation, would result in an operational profile that is no smaller than if we had pursued typical statistical testing methods. The extension method requires the construction of statistical testing and analysis machinery within our framework to generate test cases and calculate the statistics required for statistical testing analysis. Again, the benefits are a reduction in modeling effort with an added traceability from requirements to testing.

As noted earlier, an operational profile consists of structural and statistical components. The structural component already exists in the RSML^{-e}model, leaving the extension to be the addition of probabilities to the model.

3.2 Probability Assignment in RSML^{-e}

To extend a specification to include probability constructs in RSML^{-e}, we examined what probabilities to capture and how to represent them within the system model. In most flat state models, a stimulus is some user action, such as a keystroke or a menu choice. These can be modeled in RSML^{-e}, but we explore a more general solution, with respect to the capabilities of RSML^{-e}.

As noted in Section 2.2, a transition in our model is enabled when a certain condition is met. The condition's truth-value is determined through a combination of state and input variable values. At the beginning of a step, the value of the state variable is already set, so the enabling of a transition is dependent on the value of input variables that have changed. These changes are made possible in RSML^{-e} through the assignment action of input interfaces. The graph in Figure 3.1 shows how state variables are dependent on input variables, input interfaces, and the environment, in turn.

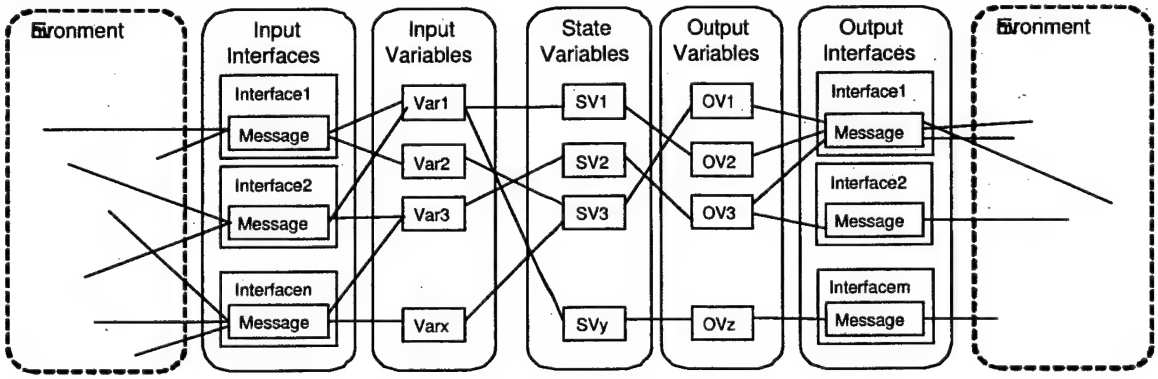

Figure 3.1: RSML^{-e} Assignment Dependency.

With this in mind, it is evident that our target for probabilities should be the observance of monitored variables at the interface with the environment. For example, in a graphical user interface, changes in the environment would be keystrokes,

mouse clicks, and entries into form fields; in more general systems we also have sensor inputs, like altitude, speed, and heading, coming over a system's interface with the environment. RSML^{-e} models these inputs as fields in messages within input interfaces. The input interface then assigns the message fields to input variables. This is demonstrated in Figure 3.2. We see that the active interface and the message fields received through that interface determine inputs to the system. We therefore have two primary probabilities to model, an interface occurrence probability and a message field value occurrence probability.

```
IN_INTERFACE UpdateInterface2 :
  MIN_SEP : UNDEFINED
  MAX_SEP : UNDEFINED
  INPUT_ACTION : RECEIVE(UpdateMsg2)
  RECEIVE_HANDLER :
    CONDITION : Var1
    ASSIGNMENT
      Var2 := f_var2,
      Var3 := f_var3
    END ASSIGNMENT
  END HANDLER

  RECEIVE_HANDLER :
    CONDITION : not(Var1)
    ASSIGNMENT
      Var2 := f_var3,
      Var3 := f_var2
    END ASSIGNMENT
  END HANDLER
END IN_INTERFACE
```

Figure 3.2: Multiple Message Handlers.

First, the interface occurrence probability indicates the probability of a particular interface being the active interface, that is, the interface that received a message. These interface occurrences are considered mutually exclusive events (according to the

semantics of RSML^{-e}), so these probabilities sum to one. Secondly, the message fields have probabilities, not to indicate the probability of occurrence, since in the interface, all message fields must have some value. Rather, these probabilities determine the value of each message field in a given message and interface.

Not only is it necessary to place probabilities on the inputs, but it is also important to allow conditional probabilities. The occurrences of many input interfaces may be dependent on the system state. For example, if a cleanroom airlock is in an idle state, the probability of getting a reset message is extremely low compared to when it is in an occupied state. It is imperative for us to be able to model this difference.

Message fields, too, have a need for conditional probabilities. In an avionics system, the probability of getting a value for altitude above a certain threshold is likely to be very high when already over the threshold and much lower if below the threshold. The modeling of these probabilities in this fashion provides the ability to create more realistic operational profiles.

Once the probabilities are recorded, they become valuable for two reasons, (1) they enable creation of statistical test cases, which consist of input messages built according to the operational profile, and (2) they are required to build the system's operational profile for statistical analysis. The question now becomes where to place these probabilities, since there are multiple locations within the RSML^{-e} model where messages are found and input variables may be assigned through multiple messages.

3.2.1 Interface Probabilities

Messages are currently referenced in two locations, the MESSAGE definition and within an IN_INTERFACE definition. Either of these could be used or a separate table could be built to maintain the probabilities of recieving messages. We chose the separate table concept and identify strengths and weaknesses of it. The other

options are discussed in Section 3.3.

The best place for the message occurrence probabilities is in a separate table, such as shown in Table 3.1. This provides a centralized location for all probabilities, allowing for checking that the sum of the message occurrence probabilities is equal to one. If changes to the probabilities are necessary, all of them can be found in one location, making for easier update, rather than having to locate each message or interface and checking the values.

Table 3.1: Input Interface Occurrence Probabilities.

Guard Condition	Input Interface Name	Probability
C1	In_Interface_1	0.30
	In_Interface_2	0.70
C2	In_Interface_1	0.40
	In_Interface_2	0.60
⋮	⋮	⋮
Cn	In_Interface_1	0.10
	In_Interface_2	0.90

This table construct provides them in a single location format, making them easy to find and update. In an RSML^{-e} model, it looks as shown in Figure 3.3.

This shows us that if condition C1 holds (could be a table or a single condition), we choose In_Interface_1 thirty percent of the time and In_Interface_2 seventy percent of the time. If C2 is true, the percents are forty and sixty, respectively. If an interface is not active under a condition, it could be given a probability of zero or be omitted.

The DEFAULT condition is allowed as an optional entry. This entry sets the probabilities if none of the prior conditions were applicable in the current state. This is further addressed in the discussion of completeness and consistency of guarding conditions, in Section 3.2.3.

```
INTERFACE_PROBABILITY_TABLE :
  CONDITION : C1
    In_Interface_1 : 0.30;
    In_Interface_2 : 0.70
  CONDITION : C2
    In_Interface_1 : 0.40;
    In_Interface_2 : 0.60
       :              :
  CONDITION : Cn
    In_Interface_1 : 0.10;
    In_Interface_2 : 0.90
  DEFAULT :
    In_Interface_1 : 0.50;
    In_Interface_2 : 0.50
END INTERFACE_PROBABILITY_TABLE
```

Figure 3.3: Input Interface Occurrence Probabilities.

3.2.2 Message Field Probabilities

In addition to the interface probabilities, we require probabilities on individual input variables, which are, in reality, the probability of occurrence of message field values in given interfaces under specific conditions. Input message fields are found in MESSAGE and IN_INTERFACE definitions, and we again use the separate table concept, such as shown in Table 3.2.

Table 3.2: Message Field Probabilities.

	Probability			
MESSAGE_FIELD Name	T	F		Guard Condition
Msg_Field_1	0.20	0.80		C1
Msg_Field_1	0.05	0.95		C2
⋮	⋮			⋮
Msg_Field_n	X 0.30	Y 0.30	Z 0.40	Cx

It would be possible to put each input message field in a table with its associated probability and any guard conditions. Again, this would make all message fields easy to find and update, if necessary. An example is shown in Table 3.2, showing both Boolean and enumerated variables. This table could be implemented as shown in Figure 3.4.

```
MESSAGE_PROBABILITY_TABLE
  UpdateInterface1 :
    CONDITION : True
      f_var1 : T [0.20], F [0.80]
  UpdateInterface2 :
    CONDITION : C1
      f_var2 : T [0.80], F [0.20];
      f_var3 : X [0.30], Y [0.30], Z [0.40]
    CONDITION : C2
      f_var2 : T [0.10], F [0.90];
      f_var3 : X [0.30], Y [0.30], Z [0.40]
END MESSAGE_PROBABILITY
```

Figure 3.4: Message Field Value Occurrence Probabilities.

The table in Figure 3.4 is similar to the one for input interfaces. However, it reads slightly differently. In this one, we know that Interface1 is active, so we check within this interface's probability definition for the correct values to assign to the message fields. This definition has one conditional, which is "True," so we assign the message fields values based on the accompanying probabilities. If C1 is true, we assign Message_Field_1 "T" twenty percent of the time and "F" eighty percent of the time. If C2 is true, those percentages become five and ninety-five, respectively. A complete listing of probabilities is shown in Figure 3.5. Again, the completeness and consistency of the guarding conditions is a concern and will be addressed in the following section.

Placing message field probabilities in a separate table makes for straightforward computation of totals and ensuring the consistency of the probabilities. This also

```
INTERFACE_PROBABILITY
  DEFAULT :
    UpdateInterface1, [0.6];
    UpdateInterface2, [0.4]
END INTERFACE_PROBABILITY

MESSAGE_PROBABILITY_TABLE
  UpdateInterface1 :
    CONDITION : True
      f_var1 : T [0.20], F [0.80]
  UpdateInterface2 :
    CONDITION : C1
      f_var2 : T [0.80], F [0.20];
      f_var3 : X [0.30], Y [0.30], Z [0.40]
    CONDITION : C2
      f_var2 : T [0.10], F [0.90];
      f_var3 : X [0.30], Y [0.30], Z [0.40]
END MESSAGE_PROBABILITY
```

Figure 3.5: Example RSML^{-e} Statistical Profile.

allows for good flexibility, since it provides the ability to add guard conditions to the message fields and it allows for easy reading and update. We also expect that it will require much simpler extensions to the syntax and parser than if the probabilities were integrated into existing entities. Finally, by placing the probabilities in a separate table or tables, the probabilities could be stored in a file separate from the rest of the model and separate files could hold different probability distributions, allowing for comparison of different probability distributions. In contrast, TML includes multiple distributions within a single model, accessed by "keys," which leads to a larger and, in our opinion, harder to read model, such as the single state with three distributions (uniform, a, and b) shown in Figure 3.6. Our method would retain the clean, readable specification model and allow the creation of one or more separate probability distributions with similar syntax and semantics.

```
[s2]
     ($ 0.8333 $)
  a: ($ 1/3 $)
  b: ($ 3/4 $)
    "Input1"     [s3]

     ($ 0.1667 $)
  a: ($ 2/3 $)
  b: ($ 1/4 $)
    "Input3"     [s5]
```

Figure 3.6: TML Multiple Probability Example.

3.2.3 Completeness and Consistency of Guard Conditions

As the probabilities are being assigned to the interfaces and message fields, it is necessary to be able to apply the same completeness and consistency checks as applied for the rest of the RSML^{-e} specification, as specified in [19]. Heimdahl and Leveson define the term "d-completeness" for specifications, which indicates the specification is complete with respect to its input domain and "consistency," where the specification has no conflicting conditions or undesired nondeterminism. We will be using the more generic term "completeness" to indicate that the specification is complete. The next state relations in RSML^{-e} are required to be a mathematical function, guaranteeing completeness, consistency, and determinism. The following paragraphs address completeness and consistency for interface occurrence probabilities and message field value occurrence probabilities individually, as they are defined differently.

Interfaces: Realistically, we expect a large set of conditions for the occurrence of interfaces, thus providing great fidelity for our modeling. These conditions must be checked for both consistency and completeness. The first of these, consistency, can be checked in the same manner as the specification is checked, by pairwise comparing

conditions to ensure mutual exclusion. Secondly, completeness can be checked in two ways, within the set of conditions and within the guarded probabilities. With a large set of conditions, it may prove to be impractical to identify every set of conditions that is not covered. Providing a default assignment of interface occurrence probabilities can solve this without creating such an exhaustive list. An example of a default interface occurrence probability assignment is shown below:

```
INTERFACE_PROBABILITY_TABLE:
  CONDITION C1 :
    In_Interface1, [0.30];
    In_Interface2, [0.70]
  CONDITION C2 :
    In_Interface1, [0.60];
    In_Interface2, [0.40]
       :
  DEFAULT :
    In_Interface1, [0.50];
    In_Interface2, [0.50]
END INTERFACE_PROBABILITY_TABLE
```

If none of the conditions applies to the current state of the model, the default set of probabilities is used to determine the interface that will be active. The second form of completeness, within each condition-guarded set of probabilities, is that the sum of the probabilities must be one. This can be seen in the above example, where each of the three sets of probabilities shown sums to one ({0.30, 0.70}, {0.60, 0.40}, and {0.50, 0.50}). The completeness check should give a warning if either of these forms of completeness is violated.

Message Fields: The message field value occurrence probabilities also must be checked for completeness and consistency. As seen in Section 3.2.2, the message field value occurrence probabilities are assigned within an interface, whose activation

will be determined elsewhere. Because the probabilities are assigned as such, it is only necessary to check completeness and consistency within this structure. For the message field value occurrence probability table in Figure 11, In_Interface1 has only one condition (True) that covers all possibilities and contains no inconsistencies for its message fields. Those fields have value occurrence probabilities of 0.20 and 0.80, which add up to one, as desired. In_Interface2 has two conditionals, Var1 and not(Var1), which are both consistent and complete. Within the not(Var1) conditional definition, var2 can be assigned based on either C1 or not(C1), a situation that is once again, both complete and consistent. By setting up the statistical profile in this manner, we have made the completeness and consistency checks simpler and enhanced the readability of these tables.

Formalization: the following section formalizes the definition of completeness and consistency for both the interface probability table and the message field value occurrence table. Given:

```
INTERFACE_PROBABILITY_TABLE:
  CONDITION C1 :
    In1, p1;
    In2, p2
  CONDITION C2 :
    In1, q1;
    In2, q2
  DEFAULT :
    In1, r1;
    In2, r2
 END INTERFACE_PROBABILITY_TABLE
```

we make the following assertions:

$$C = \{c_1, c_2, \ldots, c_n\} \quad (1)$$

$$\forall c_1, c_2 \in C, c_1 \neq c_2 \wedge (c_1 \cap c_2 = \emptyset) \quad (2)$$

$$\Sigma P[C_k] + P[DEFAULT] = 1.0, k = 1..n \quad (3)$$

$$\Sigma p_i = 1.0, i = 1..m, m = \# \text{ of interfaces} \quad (4)$$

$$\Sigma q_j = 1.0, j = 1..m \quad (5)$$

$$\Sigma r_k = 1.0, k = 1..m \quad (6)$$

The intent here is to say that no two conditions may overlap (1 and 2); the sum of all the probabilities of the conditions, including the default condition, must equal 1.0 (3); the sum of interface occurrence probabilities under a condition must sum to 1.0 (4, 5, and 6). Similarly, given:

```
MESSAGE_FIELD_PROBABILITY_TABLE
  In1 :
    v1 : val1 [p1], val2 [p2] IF True;
  In2 :
    v2 : val1 [p1], val2 [p2] IF C1;
    v2 : val1 [p3], val2 [p4] IF C2;
    v3 : val1 [p1], val2 [p2], val3 [p3] IF True;
END MESSAGE_FIELD_PROBABILITY_TABLE
```

we assert:

$$\forall \text{In}_i.v_j :$$

$$C = \{c_1, c_2, \ldots, c_n\} \quad (1)$$

$$\forall c_1, c_2 \in C, c_1 \neq c_2 \wedge (c_1 \cap c_2 = \emptyset) \quad (2)$$

$$dom(v_j) = \{val_1 val_2\} \quad (3)$$

$$\Sigma p_i = 1.0 \quad (4)$$

The intent here is to say that within a message field declaration under a given interface, no two conditions may overlap (1 and 2); all possible values for the message

field must be accounted for (3); and the probabilities within a message field declaration must sum to 1.0 (4). With these constraints in place, we can then be assured that our system will be complete, consistent, and present no nondeterminism in its implementation. This is important, especially for safety critical systems, as we must know exactly what the system will do upon any condition being met.

3.3 Design Alternatives

There are other possibilities we examined and rejected. The options are listed below, first for the interface probabilities and then for the message field probabilities, with the reasons for rejection concluding this section.

Interface Probabilities on Message Definition: It makes some sense to assign the probabilities directly on the MESSAGE definition. That would provide unit integrity, by encapsulating all message information in the MESSAGE definition. One drawback is that this limits a message probability to one value, even though the interface input it models might occur with differing probabilities depending on the current state of the system. To add multiple probabilities to the message would require adding guard conditions and probabilities to an already well-established syntax. Placing the probabilities within the message declaration also scatters the various message probabilities throughout the model, making it difficult to find all the message probabilities and ensure that they sum to one. The separate table mentioned above would keep all the message probabilities collocated, enabling easy perusal and update. An example message with two probabilities is shown below. (Note: the ## indicates lines with additions to the model.)

```
MESSAGE Update_Message {
   PROBABILITY :              ##
```

```
    TABLE                  ##
      Condition1 : 0.03;   ##
      Condition2 : 0.17    ##
    END TABLE              ##
  f_var_1 IS Boolean,
  f_var_2 IS Boolean,
  f_var_3 IS Boolean }
```

Interface Probability on Interface Definition: Message occurrence probabilities assigned directly on the input interface declarations is also an option, as all messages enter the system through some input interface. This would be a better place to assign the conditional probabilities of a message, as the interface declaration already provides the ability to define a guarding condition. One potential solution is shown in the code below. Problems that arise are the complication of the syntax and parser for the syntax, and all message occurrence probabilities again being spread throughout the model.

```
IN_INTERFACE In_Interface_Name :
  MIN_SEP : value
  MAX_SEP : value
  INPUT_ACTION : RECEIVE (In_Message_Name)
  RECEIVE_HANDLER : RCV_1
    CONDITION : C1
    PROBABILITY : real              ##
  END_HANDLER
        :
  RECEIVE_HANDLER : RCV_2
    CONDITION : C2
    PROBABILITY : real              ##
        :
  END_HANDLER
END IN_INTERFACE
```

These options could be made to work but we will use the separate table option for the advantages listed above.

Message Field Probabilities on Message Definition: Message field value assignment probabilities could be attached within the MESSAGE definition. Again, the MESSAGE definition would not be ideal, for the same reason as above, where it might be desirable to have different probabilities on a message in different interfaces or even within an interface.

```
MESSAGE Update_Message {
  f_var_1 IS Boolean
     PROBABILITY :              ##
        TABLE                   ##
          Condition1 : 0.3;     ##
          Condition2 : 0.7      ##
        END TABLE               ##
  f_var_2 IS boolean
     PROBABILITY :              ##
        TABLE                   ##
          Condition1 : 0.5;     ##
          Condition2 : 0.5      ##
        END TABLE               ##
  f_var_3 IS boolean
     PROBABILITY :              ##
        TABLE                   ##
          Condition1 : 0.7;     ##
          Condition2 : 0.3      ##
        END TABLE               ## }
```

It is evident from this example that the individual message field probabilities would be easily checked to ensure they do not sum to greater than one for any given message field. However, to change distributions would require examining, potentially, the entire model.

Message Field Probabilities on Interface Definition: As above, there is the advantage of placing different probabilities on the same message field depending on which message or interface is being used.

```
IN_INTERFACE In_Interface_Name :
 MIN_SEP : value
 MAX_SEP : value
 INPUT_ACTION : RECEIVE {In_Message_Name}
 RECEIVE_HANDLER :
   CONDITION : value
   PROBABILITY  : real                                  ##
   ASSIGNMENT
     var_1 := in_msgfld_1 [prob_msgfld_1] IF C1         ##
     var_1 := in_msgfld_1 [prob_msgfld_1] IF C2         ##
     var_2 := in_msgfld_2 [prob_msgfld_2] IF C1         ##
     var_2 := in_msgfld_2 [prob_msgfld_2] IF C3         ##
       :              :              :
     var_m := in_msgfld_m [prob_msgfld_m] IF Cx         ##
 END_HANDLER
END IN_INTERFACE
```

Advantages/Disadvantages of Design Alternatives: There are pros and cons to each of these choices. For messages, if probabilities were added in a separate table, it would make for easiest computation of totals and ensuring that the sum is not greater than one. For input message fields, probabilities in each MESSAGE would make it handy to locate all of the probabilities, but would require uniform usage in all IN_INTERFACEs in which the message field is used. By placing them in the IN_INTERFACE, it would be possible to have different probabilities of the input variable for different messages. However, message field probabilities are best placed within a separate table, just like the message occurrence probabilities. This allows for the most flexibility, since it provides the ability to add guard conditions to the message fields and it allows for easier reading and update. It also requires much simpler extensions to the syntax and parser than if the probabilities were integrated into existing entities. Finally, by placing the probabilities in a separate table or tables, the probabilities can be stored in a file separate from the rest of the model and separate files could hold different probability distributions, allowing for comparison of

different probability distributions. In contrast, TML includes multiple distributions within a model, accessed by "keys," which leads to a larger, harder to read model. Our method would leave the clean, readable specification model and create one or more easy to read probability distributions with syntax similar to that of the specification model.

Current research in the CriSys group is developing algorithms removing interfaces and converting them to state variables and identifying their message fields as input variables. This will diminish the complexity of the models and lead to smaller models for analysis. This research can be incorporated into this environment as it is finalized in the future. Little actual adjustment will be required, as the interface probabilities will be transferable to the new state variables and the message field probabilities will transfer to the new input variables. The separate table concept will make this transition smoother as well.

Chapter 4

Environmental Framework

We have shown that is is possible to extend a specification language to permit a specification model to also be an operational profile. We now describe a framework for performing specification-based testing, including both statistical and structural testing. This framework was designed to help us evaluate our statistical testing approach as well as compare random testing to structural testing. We then present the design of the test suite generator and the test execution environment for evaluating this dissertation.

4.1 Specification-based Testing

One of our goals is testing the claims of statistical testing against structural testing. Using either version under specification-based testing, we would conduct testing as shown in Figure 4.1. A specification is created to formalize the requirements of a customer. That specification is used to design and create the final implementation. The specification can also be used to generate the test cases and as an oracle to determine the expected results needed for testing the implementation.

Statistical testing follows the same plan, except that the specification is extended with a statistical profile to become an operational profile, which could be visualized as in Figure 4.2.

The results of the two forms of testing can then be compared to determine the relative

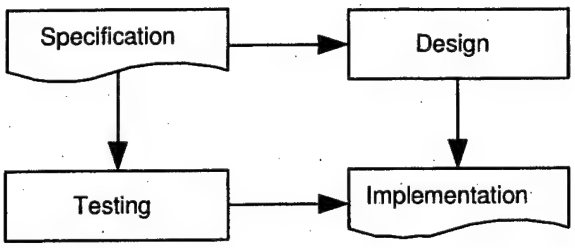

Figure 4.1: Specification-based Testing Process.

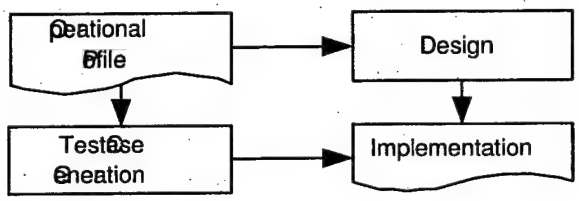

Figure 4.2: Specification-based Statistical Testing.

efficiency of the two, as shown in Figure 4.3.

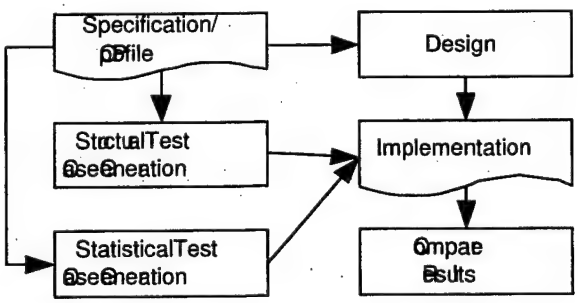

Figure 4.3: Statistical vs. Structural Testing.

4.2 Test Suite Generation and Execution

Our testing framework uses a state-based specification as the basis for an operational profile, allowing us to generate test cases and execute the test cases on the system under test. The majority of the process, illustrated in Figure 4.4, is automatable,

allowing for rapid and straightforward creation and execution of test cases.

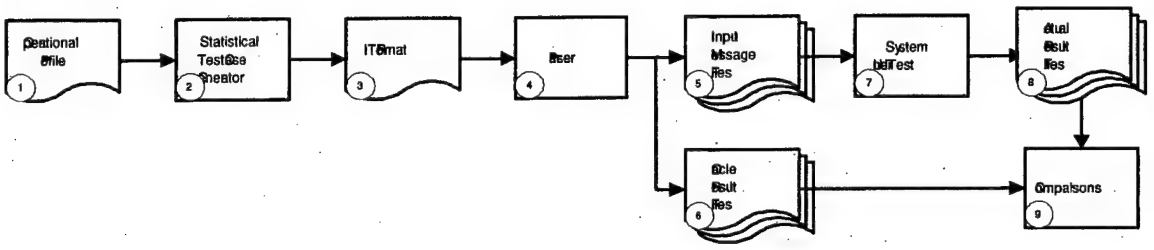

Figure 4.4: End-to-End Testing Process.

4.2.1 Test Suite Generation

The first portion, encompassing the first three blocks of the diagram, generates the test suite. The operational profile, defined in RSML^{-e}, is run through the statistical test case generator and produces a test suite in an intermediate trace representation (ITR). The system specification model, being used as an oracle, is loaded into the test case generator, the generation framework is invoked, and the generated test cases are created in the ITR format. Once the generator is invoked, the following steps take place, as shown in Figure 4.5:

1. Initialize test case counter, *tcc*, to 1.
2. Start new test case.
3. Record the initial state of the model in the test case.
4. Initialize message counter, *mc*, to 1.
5. Generate an input message from current state.
6. Record the input message in the test case.
7. Apply the input message to model.
8. Record the resulting state of the model in the test case.

9. Increment message counter. ($mc = mc + 1$)
10. If $mc <= tclen$, repeat from Step 5.
11. End test case.
12. Increment test case counter. ($tcc = tcc + 1$)
13. If $tcc <= tcnum$, repeat from Step 2.
14. Test suite complete.

We designed the ITR format for the output for the test case generator, as well as for other test case generation projects currently under development at the University of Minnesota. This format is a generic representation and incorporates the following information:

Data types. We record the data types for all input variables, to allow the test engineer to develop a testing framework that inputs valid data to the system under test.

Visible behavior of the modeled system. This includes test case inputs and expected outputs. As in traditional program testing, the expected outputs can be used as an oracle to compare with the actual output of the system or implementation under test (black-box testing).

Internal state variables. The test inputs may trigger a change in the state of the system without changing the visible behavior of the system under test. By recording the internal state variables, we can detect these changes, even though the externally visible behavior does not change.

The ITR has a declaration section and a vector of test cases output from the test case generator. The test engineer extracts individual test cases from the test case vector

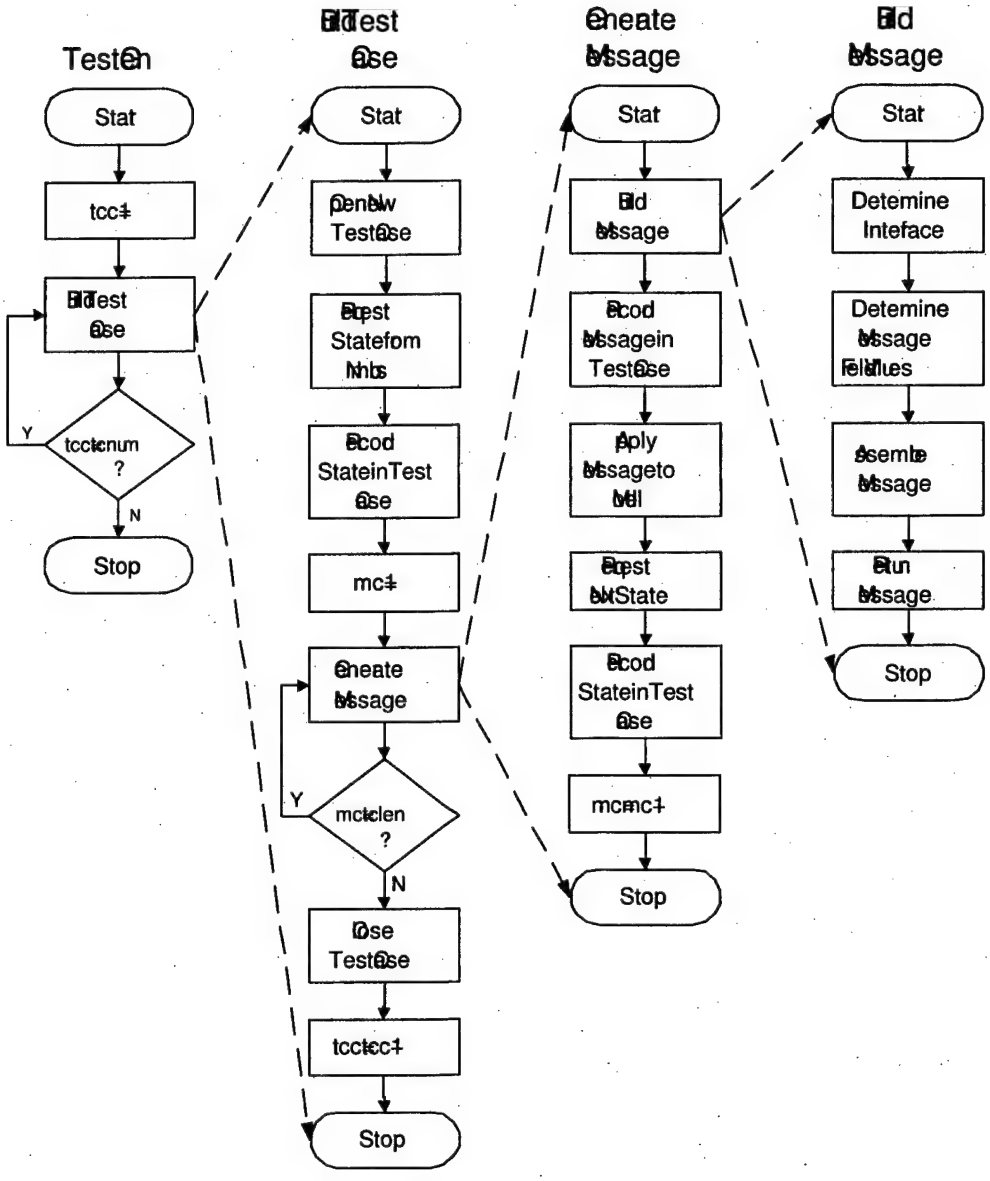

Figure 4.5: Flowchart for Statistical Testing Process.

to apply to the system under test through a test harness. The ITR is therefore a vector of test cases, each of which is a vector of steps. The test case format is shown in Figure 4.6.

```
TestCase 1
  Step 1   0:0:0:100
   /* Input Messages */
   /* State Variables */
   /* Output Variables */
  End Step 1
  Step 2   0:0:0:200
    :
  End Step 2
  :
End TestCase 1
```

Figure 4.6: Test Case Format.

Each step in a test case consists of three blocks, containing the input messages, internal state variables, and expected output variables, respectively. This format is shown in Figure 4.7. By including internal and external behavior, we provide the ability to perform white-box or black-box testing. For example, if we are testing the black-box behavior of the system, we can compare the output information with actual output and ignore the state information. Alternately, if we have access to the state of the system under test, we can perform white-box testing and check the expected state information with the actual state of the system at each step and ignore the output information.

4.2.2 Test Suite Execution

The above section is the culmination of what the research of this dissertation set out to do, namely, statistically generate test suites from a parallel and hierarchical operational profile. The remainder of Figure 4.4, blocks three through nine, is the test suite execution portion of a statistical testing framework. Regardless of what testing environment is used, the ITR format of the test suite is the starting point for test suite execution and the test engineer must build an appropriate translation

```
Step 1   0:0:0:100
   INPUT
      /* Input Messages */
      INTERFACE In_int_name : READER
         Msg_fld_name = Msg_fld_value1
         Msg_fld_name = Msg_fld_value2
               :              :
      END INTERFACE
   END INPUT

   STATE
      In_var1 = In_var_value1;
      In_var2 = In_var_value2;
       :
      St_var1 = St_var_value1;
      St_var2 = St_var_value2;
       :
   END STATE

   OUTPUT
      Out_var1 = Out_var_value1;
      Out_var2 = Out_var_value2;
       :
   END OUTPUT
End Step 1
```

Figure 4.7: Step Format.

application to convert the test suite information into the proper format for the testing environment. In general, the test suite is parsed into a translation application and the input files and oracle result files are created. Then, the system under test is executed, using the generated input files and the results of the test are recorded. In accordance with the ITR format, the results are the output variables, the internal state variables, or both. Finally, these recorded results are compared with the oracle results, indicating whether the system behaved as expected.

This framework is designed for statistical software testing. Our primary use of the framework is to show that it is possible to generate test cases statistically from complex specification models that incorporate parallelism. We also use it to compare the efficiency of statistical testing with structural testing.

Chapter 5

Case Study Overview and Related Work

Using an extended formal specification of a system to generate test cases for statistical testing provides a statistical test case generation capability that was previously unavailable. In the previous two chapters, we have shown how to extend a requirements specification language to permit the attachment of conditional probabilities on input data, based on the current values of state and input variables of the model. This permits creation of operational profiles of the system for statistical testing. We have also defined a framework for implementing statistical testing using an extended requirements specification model as an operational profile. From this definition, we have created such a testing framework and will use it for the experiments in this dissertation. However, it must be noted that we could not find nor create a valid operational profile for system under test, as neither we nor the model's creators have the capability to collect the appropriate data at present. Instead, we use a uniform distribution for the input data probabilities, resulting in a random testing strategy, also referred to as *uniform statistical testing*, a term borrowed from [62]. We use this term as it serves as a reminder that the framework is capable of true statistical test case generation even though the uniform distribution produces random testing. With our uniform statistical testing strategy, we evaluate whether our framework can generate and execute statistical test suites. With this testing framework, we also evaluate the effectiveness of generating a large number of tests, given some probabilities (in our case, uniform probabilities).

Since we now have this capability, we can use this framework to run some experiments that were previously not possible and answer questions previously unaddressed. For example,

Functionality: Can we randomly generate tests from a state-based specification model with parallel components? This allows for the testing of increasingly more complex systems.

Scalability: We would like to know how the system scales as the size of the operational profiles, as well as the size of the test cases, increases.

Effectiveness: How long should a test case be? Specifically, we look at how the length of a test case affects its effectiveness. Our system under test, an avionics component, has a very broad, flat state space, so it does not take many steps to reach the lowest levels of the system's state space. The length and number of test cases should be influential in the following areas:

1. Coverage - with a flat state space, will short or long test cases provide better coverage?
2. Fault-finding - how does the length of the test case affect the ability of the test case to detect faults?

Efficiency: How does a statistical testing strategy compare to other testing strategies? Once we determine the best length for a test case for the system under test, we can then compare our uniform statistical testing strategy against other testing strategies, cush as structural coverage testing.

From these questions, we derive the hypotheses of this dissertation. We then test these hypotheses in a case study to determine whether or not our statistical testing

framework works and compare the results of random testing with the results of a structural testing strategy in the specification domain.

Throughout these experiments, a *test case* is to be understood as a sequence of input messages for a system under test, which will guide the system from its initial state to some valid state. A *test suite* is a set of such test cases. The system will be designated as containing a *fault* if the test case terminates in a state other than the expected final state.

5.1 Testing Questions and Hypotheses

With the ability to model more complex systems, attach probabilities to the inputs from the environment, and generate test cases from the models, we can address the above questions related to statistical testing. We first run a scalability experiment to answer the first two questions above, verify (1) that the framework can generate random test cases from the operational profile and (2) that we can generate long test cases without overwhelming the test framework. This experiment is performed in conjunction with the experiments of hypotheses one and two, below.

Our first hypothesis addresses how long a test case has to be to be effective. We suspected that, due to the broad, flat state space of our system under test, a large number of short test cases would provide higher state coverage than fewer long test cases. We also suspected that, at some point, the addition of more input messages to the test cases would not provide significant increases in coverage. Under these assumptions, we selected three specification coverage criteria; state, transition, and transition decision, presented in Section 5.3.6, to evaluate the coverage capability of different length test cases. This produces the first hypothesis:

Hypothesis 1 - *Given an equivalent number of input messages, uniform statistical testing using many short test cases provides greater state, transition, and transition*

decision coverage than uniform statistical testing using fewer long test cases.

Another question that we address is how different length test cases perform at fault-finding. Much as it is expected that many short test cases will provide better coverage than few long test cases, we expect that many short test cases would be likely to find more faults than few long test cases.

Hypothesis 2 - *Given an equivalent number of input messages, uniform statistical testing using many short test cases finds more faults than uniform statistical testing using fewer long test cases.*

From the results of evaluating our first two hypotheses, we identify the most beneficial configuration for uniform statistical testing of our system under test. This configuration is then used to compare the fault-finding capability of uniform statistical testing with test suites designed to achieve various structural coverage criteria. Structural coverage testing assumes that we are likely to find more faults if we ensure that we generate a test suite to cover the entire model in some fashion, whether it is coverage of states, transitions, or some other entity. In our comparison of statistical and structural testing, we expect to be able to leverage the lack of an analysis phase in statistical testing to generate and execute significantly more test cases than structural testing in the same amount of time. Therefore, for the following two hypotheses, we hold the total effort constant, measured as the time required for generating and executing a test suite, and evaluate the two testing strategies for coverage and fault-finding capability:

Hypothesis 3 - *Given equal effort in generating and applying test cases, uniform statistical testing can provide equivalent levels of state, transition, and transition decision coverage, compared to structural coverage testing designed to provide transition decision coverage.*

Hypothesis 4- *Given equal effort in generating and applying test cases, uniform*

statistical testing can detect equivalent numbers and types of faults, compared to structural coverage testing designed to provide transition decision coverage.

These are important measures, as our long-range goal is to decrease the amount of resources required to perform testing. If statistical testing can provide equal or better fault-finding capability for equal effort, we have provided a viable alternative for software testing.

5.2 Test Infrastructure

To answer these questions and test our hypotheses, we require an infrastructure to allow the use of a specification-based operational profile as the source model for test suite generation. This infrastructure includes a source model, the testing environment, and systems to be tested.

Case Example - Flight Guidance System

For our source model, we chose a specification of the Flight Guidance System (FGS), provided by Rockwell-Collins, Inc. The University of Minnesota participates in a collaborative effort with Rockwell-Collins to model avionics systems, which has led to a collection of avionics systems models of various levels of complexity. This collection of software artifacts includes models of the FGS that range from very simple (FGS00) to the most current and complex, FGS05. The following description of the FGS is taken from [29].

The FGS is a component of the overall Flight Control System (FCS). It compares the measured state of an aircraft (position, speed, and altitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. A simplified overview of an FCS that emphasizes the role of the FGS is shown in Figure 5.1.

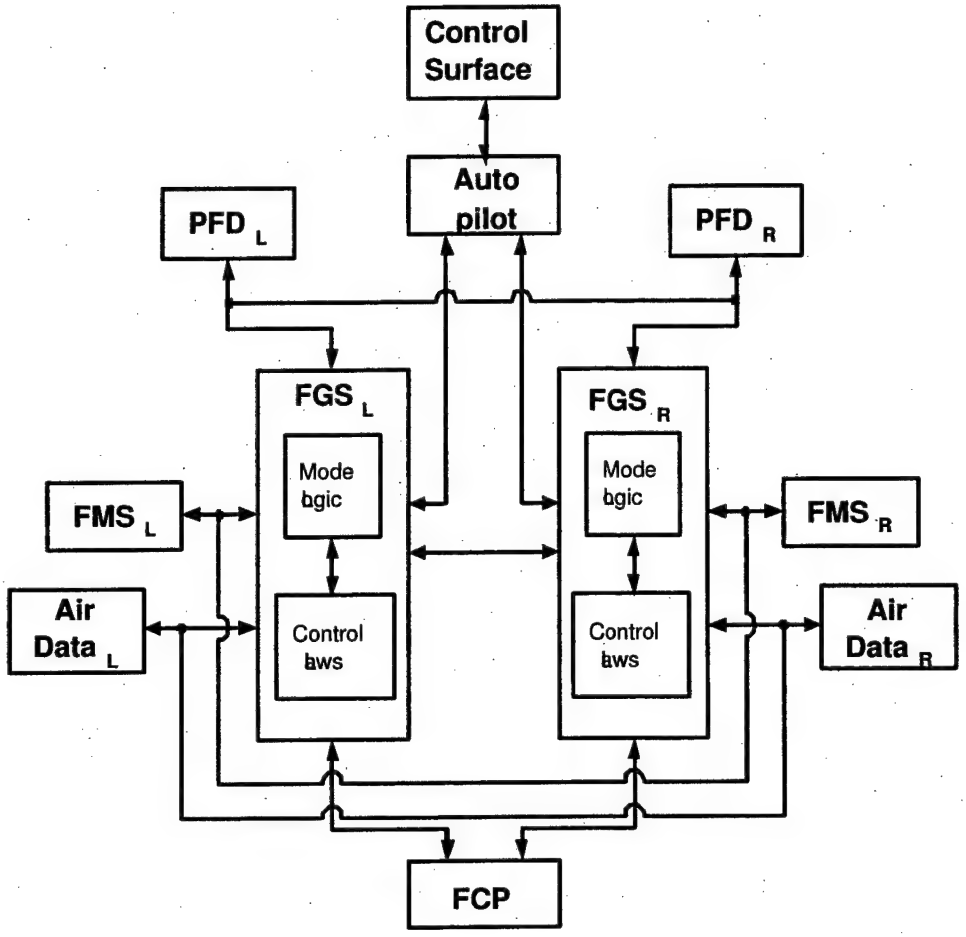

Figure 5.1: A section of the Avionics System.

The flight crew interacts with the FGS primarily through the Flight Control Panel (FCP). The FCP includes switches for turning the Flight Director (FD) on and off, and switches for selecting the different flight modes. The FCP also supplies feedback to the crew, indicating selected modes by lighting lamps on either side of a selected mode's button.

The mode logic determines which lateral and vertical modes of operation are active and armed at any given time. These in turn determine which flight control laws are active and armed. These are annunciated, or displayed, on the Primary Flight

Displays (PFD) along with a graphical depiction of the flight guidance commands generated by the FGS. The Primary Flight Display annunciates essential information about the aircraft, such as airspeed, vertical speed, attitude, the horizon, and heading. The active lateral and vertical modes are annunciated at the top of the display.

The Flight Guidance System Mode Logic

A *mode* is defined by Leveson as a *mutually exclusive set of system behaviors* [13]. The primary modes of interest in an FGS are the lateral and vertical modes. The lateral modes control the behavior of the aircraft about the longitudinal, or roll, axis, while the vertical modes control the behavior of the aircraft about the vertical, or pitch, axis. In addition, there are a number of auxiliary modes, such as half-bank mode, that control other aspects of the aircraft's behavior.

A mode is said to be *selected* if it has been manually requested by the flight crew or if it has been automatically requested by a subsystem such as the Flight Management System (FMS). The simplest modes have only two states, *cleared* and *selected.* Some modes can be *armed* to become active when a criterion is met. In such modes, the two states *armed* and *active* are sub-states of the *selected* state. Some modes also distinguish between capturing and tracking of the target reference or navigation source. Once in the active state, such a mode's flight control law first *captures* the target by maneuvering the aircraft to align it with the navigation source or reference. Once correctly aligned, the mode transitions to the *tracking* state in which it holds the aircraft on the target. Both the *capture* and *track* states are sub-states of the *active* state and the mode's flight control law is active in both states.

The *mode logic* consists of all the available modes and the rules for transitioning between them. Figure 5.2 provides an overview of the Flight Guidance System modes. Traditionally, aircraft modes are associated with a flight control law that determines

the guidance provided to the flight director or autopilot. For example, in Figure 5.2, there are lateral modes of Roll Hold, Heading Hold, Navigation, Lateral Approach, and Lateral Go Around. These control the guidance about the longitudinal, or roll, axis. Guidance about the vertical, or pitch, axis is controlled by the vertical modes of Pitch, Vertical Speed, Altitude Hold, Altitude Select, Vertical Approach, and Vertical Go Around. Each of these modes is associated with one or more control laws. In order to provide effective guidance of the aircraft, these modes are tightly synchronized. Constraints enforce sequencing of modes that are dictated by the characteristics of the aircraft and the airspace. The mode logic is responsible for enforcing these constraints.

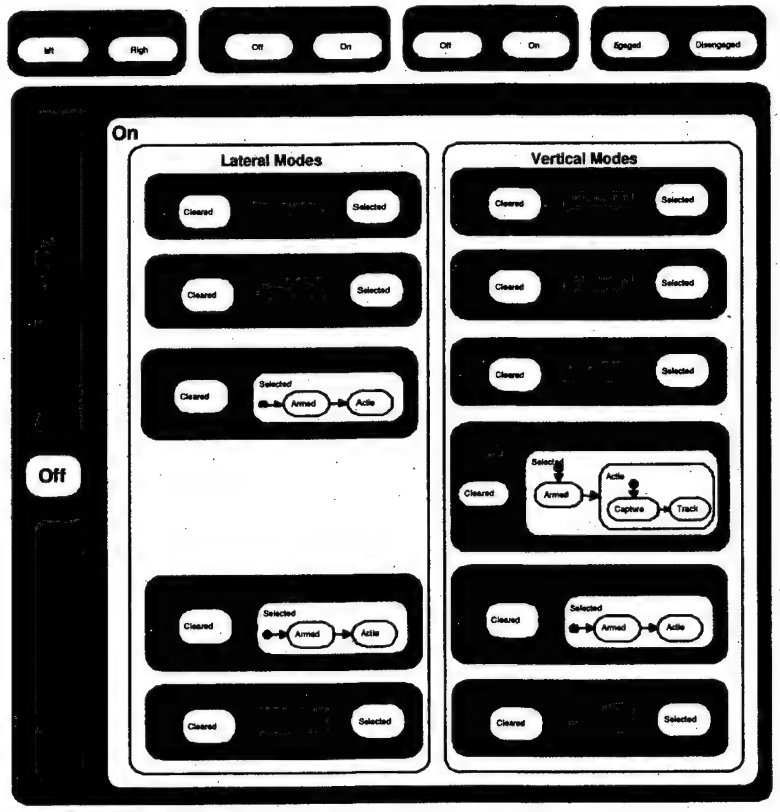

Figure 5.2: FGS Mode Logic.

The models of the FGS we use contain only Boolean and enumerated type vari-

ables. This is beneficial to our experiments, as it prevents the state space explosion problem we would encounter with infinite types and keeps the operational profile from becoming unwieldy as well.

Operational Profile

Having identified an appropriate specification for a source model, we extended it into an operational profile by adding the statistical component. As mentioned earlier, we were unable to procure a valid operational profile, so we developed an operational profile based on the uniform distribution. This uniform statistical testing produces results equivalent to random testing, making it equally likely that any message can be generated. The actual statistical component we used in our operational profile is presented in Appendix A and an abbreviated portion of it is shown here.

```
INTERFACE_PROBABILITY
   DEFAULT :
      This_Input  [0.50];
      Other_Input [0.50];
END INTERFACE_PROBABILITY

MESSAGE_PROBABILITY
   This_Input :
      DEFAULT :
         AltPreRefChanged : FALSE [0.50], TRUE [0.50];
            :                  :              :
         VsSwi             : OFF [0.50], ON [0.50]

   Other_Input :
      DEFAULT :
         AltSel      : FALSE [0.50], TRUE [0.50];
            :            :              :
         VsSel       : FALSE [0.50], TRUE [0.50]
END MESSAGE_PROBABILITY
```

In accordance with the uniform distribution, we have a default condition in the `INTERFACE_PROBABILITY` section at the top, indicating that these probabilities will be used throughout test suite generation. In this default conditional probability section, the two interfaces, `This_Input` and `Other_Input`, are each assigned a probability of 0.50, guiding the test suite generator to pick either interface with equal probability. The test case generator picks the active interface first, then uses the message probability table to determine the contents of the message to be generated. In the `MESSAGE_PROBABILITY` section, each of the interfaces is listed and, as with the interface probabilities, is given a default condition, using the default condition phrase, ensuring these probabilities are used through test suite generation. It could have been assigned using a `CONDITION : TRUE` phrase, as well. Under this default condition, we assign each message field a probability for each possible value it can assume. In this example, we have variables that are either Boolean or enumerated types with only two values, `OFF` and `ON`, so each is assigned a probability of 0.50. This gives an equal likelihood of the message field having either value assigned.

This operational profile randomly produces messages, with the intent of exercising wide portions of the system under test. The statistical testing experiments we conduct attempt to provide insight into the ability of the uniform operational profile strategy to cover the system under test.

Environment - NIMBUS Simulator

In our experiments, we use the operational profile of the FGS in a testing environment as described in Chapter 4. As we already use the NIMBUS environment for software development, we incorporated the statistical test case generator within it.

One change from the framework depicted in Figure 4.3 concerns using an implementation of a system as the system under test. The primary problem is the difficulty

of creating a manageable implementation of a system that is of significant complexity, yet has documented faults to be found. To overcome this difficulty, we use a second copy of the specification as the system under test. This specification has the same behavior as the source model specification, so we alter the behavior by seeding faults into the specification, simulating the behavior of a coded implementation. Figure 5.3 shows how a specification, seeded with faults to represent the implementation, can be tested as detailed in Chapter 4.

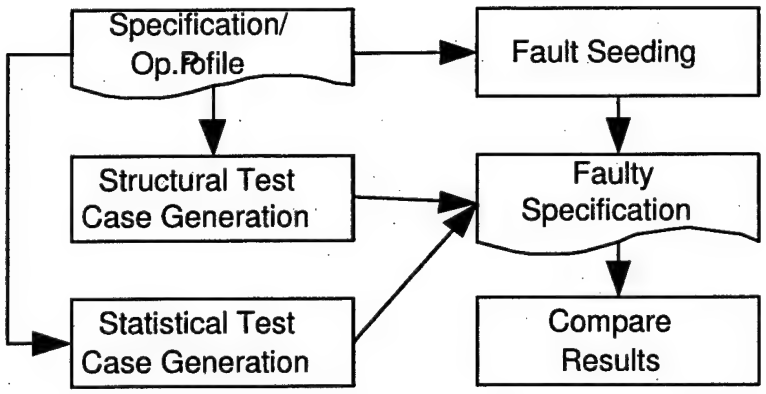

Figure 5.3: Testing with Specification as Implementation.

We can configure NimbusSim to perform as the test case generator with the correct specification and also as the implementation with the faulty specification. As shown in Figure 5.4, we store the test suite in the ITR format, load the faulty specification into NimbusSim and execute the specification with the generated input files.

The test harness reads the test cases from the file, applies the messages to the system under test, and logs the resulting state information. At the end of the testing, we check the actual state information from testing against the oracle results by comparing the state information log files with the respective oracle result files. When disagreements occur, the system records the test case and step where the error occurred, as well as identifying the disagreeing information. These results can then be

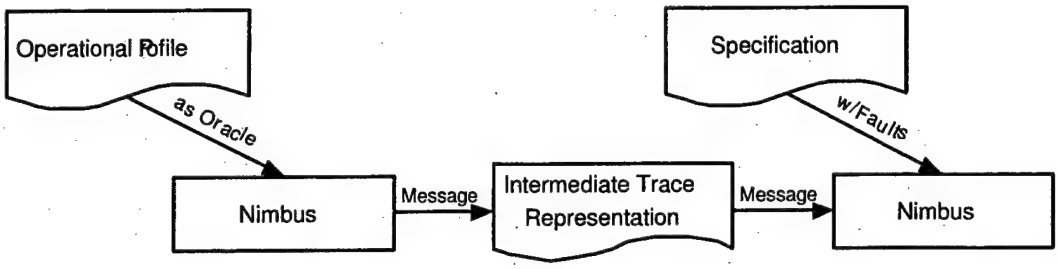

Figure 5.4: NIMBUS-to-NIMBUS Testing.

compared against the other tests we conduct to make some observations as to the efficiency of random testing.

5.3 Fault-seeding

Having created an operational profile from our requirements specification, it is necessary to have systems with known faults in order to evaluate the fault-finding capabilities of statistical and structural testing strategies. A parallel research effort at the University of Minnesota has developed an automated system for seeding several types of faults in system specifications. Here, we review related work on fault classes and mutation analysis, identify the fault types observed in real RSML^{-e} specifications, define the mutation operators we use for fault-seeding, and present our fault-seeding process. The majority of this section was developed by Jimin Gao at the University of Minnesota, to include the application in NimbusSim that generates faulty specifications.

5.3.1 Fault Classes

Previous studies have classified software faults based on Boolean formulas into four types (classes) [32, 66]:

Variable Reference Fault (VRF) - a Boolean variable x is replaced by another variable y, where $x \neq y$.

Variable Negation Fault (VNF) - a Boolean variable x is replaced by $\overline{x}$.

Expression Negation Fault (ENF) - a Boolean expression p is replaced by $\overline{p}$.

Missing Condition Fault (MCF) - a failure to check a precondition.

It has been shown that, for test vector generating purpose, $S_{VRF} \Rightarrow S_{VNF} \Rightarrow S_{ENF}$ and $S_{MCF} \Rightarrow S_{VNF} \Rightarrow S_{ENF}$, where S represents the test conditions for detecting these faults. Namely, a test vector that can detect a variable reference fault or missing condition fault for a Boolean variable occurrence can also detect its variable negation fault and expression negation fault for the expression containing this occurrence. This relation may be used to reduce the size of the test suite; it can also be used to reduce the number of faulty systems being tested, as subsumption identifies duplication or redundancy. However, we are more interested in the relative efficiency of various testing strategies, not the absolute efficiency of them. Therefore, we will generate faulty systems for testing without regard to potential duplication of faults.

Similar fault classes were identified in [61] and [43] but in relation to faults found in programs, not specifications.

5.3.2 Mutation Analysis

Mutation analysis, combined with model checking, has been used to generate test cases for safety property testing [3]. Mutation operators, applied to original expressions to generate mutant specifications, were further studied in [4]. The mutation operators relevant to this case study are:

Operand Replacement Operator (ORO) - replace an operand (a variable or constant) with another syntactically legal operand.

Single Expression Negation Operator (SNO) - replace a simple expression (a Boolean variable or an expression in the form *token*1 *operator token*2, where *token*1 and *token*2 are variables of scalar type or constants, and *operator* is a relational operator) by its negation.

Expression Negation Operator (ENO) - replace an expression with its negation.

Logical Operator Replacement (LRO) - replace a logical operator with another logical operator.

Relational Operator Replacement (RRO) - replace a relational operator with another relational operator, except for its opposite.

Missing Condition Operator (MCO) - delete a simple expression from a Boolean expression.

Stuck-At Operator (STO) - consists of two operators. Stuck-At-0 replaces a simple expression with 0 and Stuck-At-1 replaces a simple expression with 1.

Associative Shift Operator (ASO) - change the association between variables.

These mutation operators generally do not correspond exactly to the fault classes discussed in [32]. However, ORO combined with RRO generates a class of faults closely matching VRF.

Although these mutation operators correspond to a more general set of fault types than those discussed above, in this dissertation they are focused on Boolean expressions.

5.3.3 Representative Faults in RSML^{-e}

As mentioned earlier, we possess a library of past versions of the FGS specification. The revision history of the FGS05 specification were examined in order to get an idea of what types of faults are likely to appear in real RSML^{-e} models. The issues we discovered are:

1. Misspecified conditions. This can be in either a macro or state transition condition.

   (a) Missing conditions: in later versions, new predicates were added to an AndOrTable, or a truth value of * was changed to `T` or `F`.

   (b) Redundant or unnecessary conditions: in later versions, conditions were deleted from an AndOrTable, or a truth value of `T` or `F` was changed to *.

   (c) Condition negation error: in later versions, a truth value of `T` was changed to `F` or a truth value of `F` was changed to `T`.

2. Incorrect initial values. As an example, the initial value of a state variable or input variable was changed in a later version from `Undefined` to `True`.

3. Variable reference error. A misuse of macro or variable names, such as `Is_LAPPR_Active` where `Is_LAPPR_Selected` should be used.

Except for the incorrect initial values, most faults appear in AndOrTables, the counterpart of a Boolean condition in other specification languages. The incorrect initial values can be viewed as a special type of operand replacement fault that is unique to RSML^{-e} .

5.3.4 Mutation Operators

Considering the mutation operators and the fault types discussed above, mutation operators tailored for RSML^{-e} can be defined and applied to generate faulty specifications. Because RSML^{-e} enforces the use of AndOrTables for Boolean conditions by not providing the `AND` and `OR` operators, some fault types, such as the associative shift faults and the logic operator replacement faults, cannot appear in a RSML^{-e} specification. They are therefore omitted from our study. Below we show the proposed mutation operators, their justifications, and illustrating examples.

Variable Replacement Operator: Replaces a variable, macro or constant name reference with names of the same type. When applied to AndOrTable predicates, incorrect variable references will be generated. This covers the fault issue #3, one of most frequently seen fault types in RSML^{-e} . For example, the following AndOrTable

```
TABLE
  When_HDG_Activated()     : T * * *;
  When_NAV_Activated()     : * T * *;
  When_LAPPR_Activated()   : * * T *;
  When_LGA_Activated()     : * * * T;
END TABLE
```

can be changed to

```
TABLE
  When_HDG_Activated()     : T * * *;
  When_NAV_Activated()     : * T * *;
  Is_LAPPR_Active()        : * * T *;
  When_LGA_Activated()     : * * * T;
END TABLE
```

Condition Insertion Operator: Replaces a truth value in an AndOrTable, in either a macro or state transition condition, from * to T or F. This covers fault type 1b. For example, the AndOrTable

```
TABLE
  When_HDG_Activated()     : T * * *;
  When_NAV_Activated()     : * T * *;
  When_LAPPR_Activated()   : * * T *;
  When_LGA_Activated()     : * * * T;
END TABLE
```

can be changed to

```
TABLE
  When_HDG_Activated()     : T * * *;
  When_NAV_Activated()     : * T * *;
  When_LAPPR_Activated()   : * * T T;
  When_LGA_Activated()     : * * * T;
END TABLE
```

Condition Removal Operator: Replaces a truth value in an AndOrTable, in either a macro or state transition condition, from T or F or *. This covers fault type 1a. For example, the AndOrTable

```
TABLE
  When_HDG_Activated()     : T * * *;
  When_NAV_Activated()     : * T * *;
  When_LAPPR_Activated()   : * * T *;
  When_LGA_Activated()     : * * * T;
END TABLE
```

can be changed to

```
TABLE
  When_HDG_Activated()     : T * * *;
  When_NAV_Activated()     : * T * *;
```

```
  When_LAPPR_Activated()   : * * * *;
  When_LGA_Activated()     : * * * T;
END TABLE
```

Condition Negation Operator: Replaces a truth value in an AndOrTable, in either a macro or state transition condition, from `T` or `F` or from `F` to `T`. This covers fault type 1c. For example, the AndOrTable

```
TABLE
  When_HDG_Activated()     : T * * *;
  When_NAV_Activated()     : * T * *;
  When_LAPPR_Activated()   : * * T *;
  When_LGA_Activated()     : * * * T;
END TABLE
```

can be changed to

```
TABLE
  When_HDG_Activated()     : T * * *;
  When_NAV_Activated()     : * T * *;
  When_LAPPR_Activated()   : * * F *;
  When_LGA_Activated()     : * * * T;
END TABLE
```

Literal Replacement Operator: A literal value occurrence, in constant definitions, state variable or input variable initial value declarations, state transition target value or source value expressions, or AndOrTable predicates, is replaced with another value of the same type or `Undefined`. This covers fault type 2. For example, the following state variable definition

```
STATE_VARIABLE Is_ROLL_Selected: Boolean
   PARENT : NONE
   INITIAL_VALUE : FALSE
   CLASSIFICATION: CONTROLLED

   EQUALS (..ROLL = Selected) IF TRUE
END STATE_VARIABLE
```

can be changed to (two literals are changed for illustrative purpose)

```
STATE_VARIABLE Is_ROLL_Selected: Boolean
   PARENT : NONE
   INITIAL_VALUE : TRUE
   CLASSIFICATION: CONTROLLED

   EQUALS (..ROLL = Cleared) IF TRUE
END STATE_VARIABLE
```

and as another example, the constant definition

```
CONSTANT THIS_SIDE : Side
   VALUE : LEFT
END CONSTANT
```

can be changed to

```
CONSTANT THIS_SIDE : Side
   VALUE : RIGHT
END CONSTANT
```

Stuck-at Operator: An AndOrTable, in a macro definition or in a state variable transition condition, is replaced by `True` (stuck-at-true) or `False` (stuck-at-false). Although these types of faults are less likely to appear in real RSML^{-e} specifications, they potentially can be used to evaluate the quality of the test suite. For example, the macro definition

```
MACRO Overspeed_Condition() :
   TABLE
      Overspeed != UNDEFINED  : T;
      Overspeed = TRUE        : T;
   END TABLE
END MACRO
```

can be changed to

```
MACRO Overspeed_Condition() :
   TRUE
END MACRO
```

for a stuck-at-true fault.

Predicate Removal Operator: A row in an AndOrTable is removed, resulting in single or multiple missing condition faults. For example, the AndOrTable

```
TABLE
   When_ALT_Switch_Pressed_Seen()                : T *;
   PREV_STEP(Is_VAPPR_Active)                    : F F;
   When_ALTSEL_Target_Altitude_Changed_Seen()    : * T;
   PREV_STEP(Is_ALTSEL_Track)                    : * T;
END TABLE
```

can be changed to

```
TABLE
   When_ALT_Switch_Pressed_Seen()                : T *;
   When_ALTSEL_Target_Altitude_Changed_Seen()    : * T;
   PREV_STEP(Is_ALTSEL_Track)                    : * T;
END TABLE
```

These types of operations are simply synthetic effects of several truth value replacement operations. They are needed however, because people do make these mistakes when writing specifications.

NOT Operator Removal/Insertion Operator: A `NOT` operator is removed from or inserted into a Boolean expression. `NOT` is the only logic operator allowed in RSML^{-e} . It can appear in the AndOrTable predicates, in state variable transition conditions or macro definitions if an AndOrTable is not used, and in state variable transition target value expressions. The removal or insertion of a `NOT` in AndOrTable predicates has the same effect as a condition negation

operation, so this operator should be used only for full condition expressions and target value expressions. For example, the state variable definition

```
STATE_VARIABLE Independent_Mode: On_Off
   PARENT : None
   INITIAL_VALUE : Off
   CLASSIFICATION: State

   EQUALS On IF Independent_Mode_Condition()
   EQUALS Off IF NOT Independent_Mode_Condition()
END STATE_VARIABLE
```

can be changed to

```
STATE_VARIABLE Independent_Mode: On_Off
   PARENT : None
   INITIAL_VALUE : Off
   CLASSIFICATION: State

   EQUALS On IF NOT Independent_Mode_Condition()
   EQUALS Off IF Independent_Mode_Condition()
END STATE_VARIABLE
```

(Note: Both removal and insertion were used for illustrative purposes.)

Relational Operator Replacement Operator: If its operands are of `Integer` or `Real` type, a relational operator is replaced with another relational operator except for its opposite. This operator generates an incorrect predicate with a mistreated boundary value. For example, the macro definition

```
MACRO AboveThresholdHyst() :
   TABLE
      AltitudeQ1 = Good                              : T F T;
      Altitude1 = UNDEFINED                          : F * F;
      Altitude1 > AltitudeThreshold + Hysteresis : T * T;
      AltitudeQ2 = Good                              : F T T;
```

```
      Altitude2 = UNDEFINED                          : * F F;
      Altitude2 > AltitudeThreshold + Hysteresis : * T T;
   END TABLE
END MACRO
```

can be changed to

```
MACRO AboveThresholdHyst() :
   TABLE
      AltitudeQ1 = Good                              : T F T;
      Altitude1 = UNDEFINED                          : F * F;
      Altitude1 > AltitudeThreshold + Hysteresis : T * T;
      AltitudeQ2 = Good                              : F T T;
      Altitude2 = UNDEFINED                          : * F F;
      Altitude2 >= AltitudeThreshold + Hysteresis: * T T;
   END TABLE
END MACRO
```

However, since relational expressions are rare in RSML^{-e} specifications (nonexistent in the FGS models), this operator is less important at this stage and it is not implemented here.

Numeric Operator Replacement Operator: A numeric operator in AndOrTable predicates or in state transition target value expressions can be replaced by another numerical operator. For example, the macro definition shown above can be changed to

```
MACRO AboveThresholdHyst() :
   TABLE
      AltitudeQ1 = Good                              : T F T;
      Altitude1 = UNDEFINED                          : F * F;
      Altitude1 > AltitudeThreshold + Hysteresis : T * T;
      AltitudeQ2 = Good                              : F T T;
      Altitude2 = UNDEFINED                          : * F F;
      Altitude2 > AltitudeThreshold - Hysteresis : * T T;
   END TABLE
END MACRO
```

For the same reason as discussed above, the implementation of this operator is not necessary at this stage.

For our experiments, we seed only the first four fault types—variable replacement, condition insertion, condition negation, and condition removal. These are the ones that were most prevalent in our review of the revision histories of the FGS model. Also, some fault types, such as the relational and numeric operator replacement faults cannot occur in the FGS, as it does not contain these operators. The four we have chosen provide a good target for our experiments.

5.3.5 Fault Seeding

As noted earlier, Jimin Gao of the University of Minesota created the fault seeder used for generating faulty specifications for this dissertation. The fault seeder produces multiple faulty versions of a specification, each with a single fault of a specified type. The fault seeder is implemented in NIMBUS using the visitor design pattern [14] and makes controlled transformations to the internal representation of the specification, an abstract syntax tree (AST). The fault seeder has the following components, as illustrated in Figure 5.5:

1. The *User-Interface* component queries the fault type to be seeded and number of faulty versions to be generated for the subject specification.

2. The *Fault Seeder* performs the fault seeding by picking the replacements corresponding to that fault type at random using a random number generator. It also annotates information regarding the location of fault injection and the modification performed.

3. The *Reverse-Specification Generator* takes the mutated AST and translates it

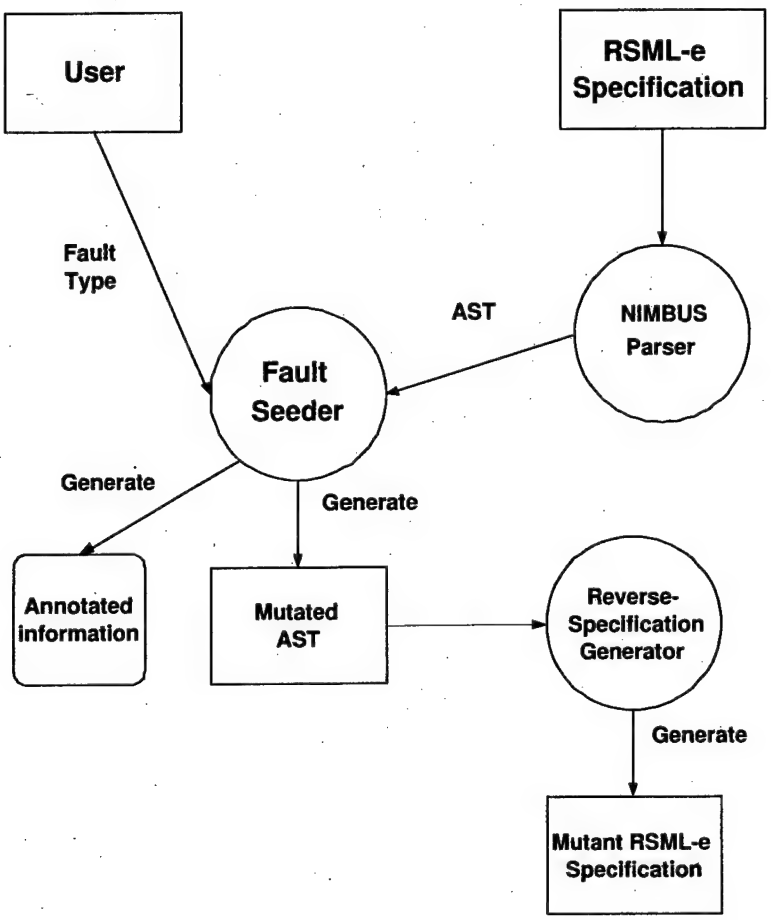

Figure 5.5: Fault Seeder in NIMBUS

back to the specification in RSML^{-e} , resulting in a mutant RSML^{-e} specification.

We place the fault seeder into our testing process, as shown in Figure 5.6. This results in our having a source model from which to generate tests, an ability to generate specifications with known faults to test, and a framework for testing. At this point we can discuss the test quality measures we will use in our experiments.

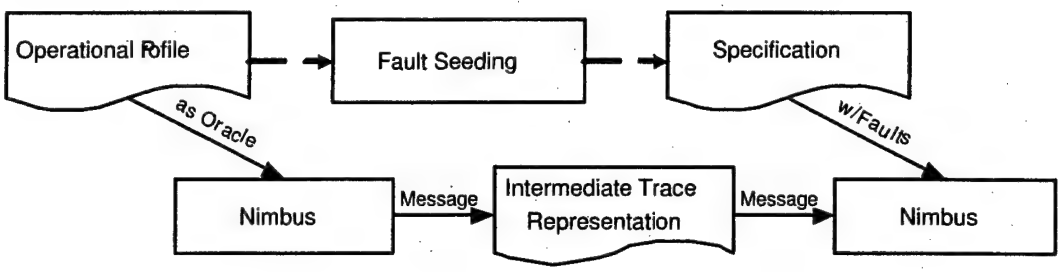

Figure 5.6: NIMBUS-to-NIMBUS Testing with Fault Seeding.

5.3.6 Test Coverage Measures

Now that the testing framework is complete, with an oracle and mutant systems to test, we can look at the analysis portion of our experiments. Various test coverage metrics have been created and applied to models, including, in decreasing coverage, all execution paths, all definition-use paths, all definition paths, all predicates, all edges and all nodes [28]. Some related coverage metrics have been defined in [50] and [41], to be used in evaluating specification models, such as those created in RSML^{-e}, that may possess parallel components. These include state coverage, transition coverage, transition decision coverage, column coverage, and clause-wise guard coverage. The first three of these are the coverage criteria we use in our comparison experiments.

State coverage: *A test suite is said to achieve state coverage of a state variable in an RSML^{-e} specification if, for each possible value of the state variable, there is at least one test case in the test suite that assigns that value to the given variable. The test suite achieves state coverage of the specification if it achieves state coverage for each state variable.*

State coverage is analogous to all nodes coverage in customary data flow diagram evaluations. In state coverage of a state variable, test cases exist to make the state variable take on each possible value at least once. The test suite ensures state coverage of the specification by achieving state coverage of each state

variable. This would be the minimum level of coverage testing we would expect to be able to perform for comparison.

Metric: Given a set of test cases T, the state coverage obtained by executing the test cases on the RSML^{-e} specification can be measured as follows:

State coverage of the specification by the set of test cases T =

$$[\sum_{i=1}^{N} value\_mark(s_i) / \sum_{i=1}^{N} value\_size(s_i)] * 100$$

where N is the number of state variables defined in the model, $value\_mark(s_i)$ is the number of variable values marked in state variable i, and $value\_size(s_i)$ is the number of possible values state variable i can take, including undefined.

Transition Coverage: *A test suite is said to achieve transition coverage of a state variable in an RSML^{-e} specification if, for each specified transition of the state variable, there is a test case in the test suite that takes that transition.*

Transition coverage is similar to state coverage, except it addresses the transitions out of a state, rather than the states themselves. A test suite is said to provide transition coverage if it requires that the test cases ensure that each guard (table of conditions or single condition) takes on a true value at some point within the test suite.

Metric: Given a set of test cases T, the transition coverage obtained by executing these test cases on the RSML^{-e} specification can be measured as follows:

Transition coverage of the specification by the set of test cases T =

$$[(\sum_{i=1}^{N} t[s_i]) / (\sum_{i=1}^{N} c[s_i])] * 100$$

where N is the number of state variables in the model, s_i is state variable i,

$t[s_i]$ is the number of true markers for transitions in s_i, $c[s_i]$ is the number of transitions (cases) for state variable i.

Transition Decision Coverage: *A test suite is said to achieve transition decision coverage of a given RSML^{-e} specification if each guard condition (specified as either an AND/OR table or as a standard Boolean expression) evaluates to true at some point in some test case and evaluates to false at some point in some other test case in the test suite.*

Transition decision coverage is similar to transition coverage, except it requires that the test cases ensure that each guard (table of conditions or single condition) takes on both true and false values within the test suite. This is equivalent to all predicates testing and gives a more complete coverage of the model than state coverage.

Metric: Given a set of test cases T, the transition decision coverage obtained by executing these test cases on the RSML^{-e} specification can be measured as follows:

Transition decision coverage of the specification by the set of test cases T =

$$[\sum_{i=1}^{N}(t[s_i] + f[s_i])/(2 * \sum_{i=1}^{N} c[s_i])] * 100$$

where N is the number of state variables in the model, s_i is state variable i, $t[s_i]$ is the number of true markers for transitions in s_i, $f[s_i]$ is the number of false markers for transitions in s_i, and $c[s_i]$ is the number of transitions corresponding to the state variable i (this last sum is multiplied by two to account for both true and false possibilities in each case).

These three test coverage measures are implemented in the NIMBUS simulator and will be used for evaluating the efficiency of statistical and structural testing.

5.4 Structural Test Suite Generation

We will use these test coverage measures to generate test suites for comparison with statistical test suites. These test suites can be generated using a model checker. We will not get into the technical details of how this is achieved since it is outside the scope of this dissertation. The interested reader is refered to Dr. Sanjai Rayadurgam's dissertation [49] and the following publications [20, 50, 51, 52, 53]. The following brief discussion of the general approach has been adapted from [51].

Model checkers build a finite state transition system and exhaustively explore the reachable state space, searching for violations of the properties under investigation [11]. Should a property violation be detected, the model checker will produce a counter-example illustrating how this violation can take place. In short, a counter-example is a sequence of inputs that will take the finite state model from its initial state to a state where the violation occurs.

A model checker can be used to find test cases by formulating a test criterion as a verification condition for the model checker. For example, we may want to test a transition (guarded with condition C) between states A and B in the formal model. We can formulate a condition describing a test case testing this transition—the sequence of inputs must take the model to state A; in state A, C must be true, and the next state must be B. This is a property expressible in the logics used in common model checkers, for example, the logic LTL. We can now challenge the model checker to find a way of getting to such a state by negating the property (saying that we assert that there is no such input sequence) and start verification. We call such a property a *trap property.* The model checker will now search for a counterexample demonstrating that this trap property is, in fact, satisfiable; such a counterexample constitutes a test case that will exercise the transition of interest. By repeating this process for each transition in the formal model, we use the model checker to

automatically derive test sequences that will give us transition coverage of the model. Obviously, this general approach can be used to generate tests for a wide variety of structural coverage criteria, such as all state variables have taken on every value, and all decisions in the model have evaluated to both true and false.

The approach discussed above is not unique to our group; several research groups are actively pursuing model checking techniques as a means for test case generation [2, 3, 15, 25, 50]. Nevertheless, to our knowledge, no experimental data is available about the fault finding capability of test suites automatically generated to various specification coverage criteria.

We have a source model to use for statistical testing, we can attach probabilities to it to make it an operational profile for generating test cases, we have specifications to be tested which we have seeded with faults, we can generate structural test suites, we can generate statistical test suites, we have a framework in which to run it all, and we have the ability to create structural criteria-based test suites to compare to. Given these capabilities, we are in a position to design and execute experiments to evaluate the capabilities of statistical testing and compare a statistical testing strategy against a structural testing strategy.

Before we continue discussing the details of our experiments and the experimental results, we will provide a discussion of related work comparing various testing strategies.

5.5 Related work

We found only a few cases of testing research directed at specifications using statistical or random testing or using a state-based specification model as the basis for an operational profile. The experiments we found could be divided into two main categories, comparative studies and analytical studies; [75] provides greater delineation

of testing strategies and experiments by breaking comparative studies into statistical experiments and simulation. We will address these both as comparative studies, as we found them used together in several experiments.

5.5.1 Comparative studies

Various comparative testing experiments have been conducted for testing strategies, such as coverage testing against statistical testing and random testing. Most closely related to our work are experiments by Offutt, *et al.*, based on test generation from Unified Modeling Language (UML) state diagrams [39, 40, 41, 42]. In these experiments, Offutt aims to "provide a solid foundation for generating tests from system level specifications" in order to achieve higher assurance of quality and reliability in software systems from the avionics, medical, and other control systems domains. He uses UML state diagrams to generate test case data to detect seeded faults in an implementation. This is very similar to the work of this dissertation, except we use an RSML^{-e} specification as an operational profile to generate test suites and will execute our test suites on fault-seeded specifications, not implementations. Also, Offutt uses a deterministic process to produce the test cases, where we use a probabilistic process.

For their testing, Offutt defines four structural coverage criteria, as follows:

Transition: *The test set must satisfy every transition in the specification graph.* This is essentially the same as our definition.

Full Predicate: *For each predicate P on each transition, the test set must include tests that cause each clause c in P to result in a pair of outcomes where the value of P is directly correlated with the value of c.* This tests each clause in each predicate, where a clause is a Boolean expression without any of the Boolean operators AND, OR, and NOT.

Transition-Pair: *For each pair of adjacent transitions $S_i : S_j$ and $S_j : S_k$ in the specification graph, the test set contains a test that traverses the pair of transitions in sequence.* This level of coverage tests all pairs of transitions rather than just single transitions.

Complete Sequence: *The test engineer must define meaningful sequences of transitions on the statechart diagram by choosing sequences of states that should be entered.* A complete sequence is a sequence of state transitions that form a complete usage of the system and this coverage measure is equivalent to the all-execution paths coverage definition [28]. Coverage to this level is impractical in many cases, as the number of complete sequences for a realistic system is often infinite. Realistically, a test engineer is required to develop a test set for this coverage measure.

The UML state diagrams are converted into test cases using a test specification language, which specifies pre-state and post-state information, input values, and control commands. Offutt generates test suites to provide transition-pair and full predicate coverage and then applies them to the implementations under test. For comparison purposes, a statement coverage test suite was hand crafted by persons not involved directly in the study.

In the experiments of [39], Offutt uses thirty-four full predicate test cases and thirty-four transition-pair test cases on twenty-five single-point mutants of a cruise control model and plots the results as the number of mutants killed by coverage type. As shown in Table 5.1, the full predicate test suite found all twenty-five faults, the transition-pair test suite found eighteen faults, and the statement coverage test suite found sixteen.

In [40], similar experiments are conducted, again against the cruise control model. In these experiments, they generated fifty-four full predicate test cases, thirty-four

Table 5.1: Testing Results - Offutt.

Test Suite	Faults Found	Percent Found
Full Predicate	25	100
Transition-pair	18	72
Statement	16	64

transition-pair test cases, twelve transition test cases, and fifty-four random test cases for comparison. The test sets were executed on twenty-four mutant implementations of the cruise control, each with one seeded fault. These faults were variable replacement faults, arithmetic operator faults, and four "naturally occurring" faults. Again, the full predicate test set performed best finding twenty of the twenty-four seeded faults. As shown in Table 5.2, the transition-pair test set found eighteen, the transition test set found fifteen, and the random test set found fifteen.

Table 5.2: Testing Results - Offutt #2.

Test Suite	Faults Found	Percent Found
Full Predicate	20	83
Transition-pair	18	75
Transition	15	63
Random	15	63

In both of these experiments, full-predicate coverage testing performed well and better than any other strategy they used. Although full-predicate testing did not find all the faults in the second set of experiments, Offutt offers an explanation. Offutt notes that the testing revealed a flaw in their process, where the full predicate coverage could not find faults if the test engineer sets the pre-state to place the system in a certain state, then runs the test set. The explicitly set pre-state information bypasses the fault and prevents its detection. If the pre-state information is included as part of the test set, it uses inputs to set the state and the fault is found. This accounts

for three of the four missed faults; the fourth was not found by any of the testing, but no explanation was provided for it.

Chevalley's [6] experiments are closely related to Offutt's, except they impose a probability distribution over the input domain to produce random test cases from a UML state diagram. The goal is to devise tests from object-oriented specifications. Offutt's process requires one pass through an element for coverage, whereas Chevalley's ensures that each element is exercised several times. This process is called *statistical functional testing*, a combination of *statistical testing*, which focuses on probabilistic choice of input values, and *functional testing*, which focuses on a functional model of the system.

Two factors are considered to produce a test case in this process, the input probability distribution and the test size. Determining the input distribution is the key component of the process and focuses on a given element of the model. Given a testing criterion, such as transition coverage, S is the set of transitions and P is the occurrence per execution of the least likely element (transition) of S. The solution to the test set size then becomes an optimization problem that can be solved analytically or empirically. The calculation of the occurrence per execution of the least likely element is essentially the same as the analysis performed by JUMBL [46]. The probability distribution is then developed to exercise the least likely transition the desired number of times. Chevalley found that the semi-automated solution of the optimization problem was tedious if the transitions' preconditions are highly dependent on each other. They proceeded with an empirical determination of the input distribution. This determination consisted of running a random distribution of input values, recording the occurrence of the elements, then adjusting the distribution to cover the least covered elements. Several iterations of this process are necessary to develop the input distribution used in testing. Upon attempting to determine their

input distribution, they quickly learned that most inputs from the input domain did not cause changes in variables. They altered their algorithm to pick input values that would force changes in variables that led to transitions being exercised. A computer-aided software engineering (CASE) tool was required to provide programmable model execution and recording of results, including the coverage measures. Even with an automated system, they discovered that "balanced" coverage–exercising all elements equally–was not possible, as the least covered elements required more highly covered elements to be triggered. To exercise the least covered elements the given number of times, they were forced to accept this condition.

They test an FGS model and its Java implementation, provided by Rockwell-Collins. For the feasibility study, they used only four UML state diagrams with twelve transitions and eleven input variables of either Boolean or float type. The coverage measure used for generating the tests was transition coverage, defined like ours and Offutt's.

They ran five sets of inputs against forty-six mutants provided by Offutt, using both random and functional testing. They considered a mutant killed if it was killed by all five test sets. The following table, Table 5.3, shows how the number of mutants killed levels out around 300 test inputs for both strategies. The statistical functional test suite performed well, killing 41 of 46 mutants, compared to 32 by random testing using the uniform distribution.

Table 5.3: Testing Results - Chevalley.

Test Size	Uniform	Functional
100	16	31
200	28	41
300	31	41
400	32	41
500	32	41

The five mutants not killed by all five test sets were the result of changes in constant values. The bands of input values that could discover these faults were extremely small (e.g., "$c < 0.813$" replaced with "$c < 0.815$" has a band of [0.813,0.815]).

They noted that their method would add an order of magnitude in effort and complexity (and is possibly even infeasible) when addressing coverage measures more stringent than transition coverage, so more research is needed to address this issue. They concluded that statistical functional generation of input data from a UML diagram is effective for killing mutants. Their experiments also indicated that ninety-five percent of seeded faults found could be a valuable stopping measure. They noted that the amount of testing required to find faults beyond the ninety-five percent level was no longer cost-effective.

Thevenod-Fosse, *et al.* [62], ran comparative experiments executing eighty-two test sets against mutated implementations of four C programs, using *structural deterministic testing*, the same as our definition of structural testing, and *structural statistical testing*, which imposes a random distribution over the input domain. The goal of these experiments was to compare deterministic and random testing. They generate seventy-three structural deterministic test sets and apply them to 2816 mutant implementations. Similarly, they generated 9 statistical test sets to apply to the mutants. The efficiency of the test sets were judged as the number of killed mutants divided by the total number of mutants. They discovered that their deterministic testing never scored higher than the statistical testing and, in fact, deterministic testing varied quite widely, based on the inputs chosen. They determined that deterministic testing can perform poorly, since it selectively chooses a small set of input test data, which may not be adequate for fault-finding. They also determined that the uniform statistical distribution performs worse as the system structure lends itself less toward a statistical distribution, that is, if the fault is reachable through a small

set of inputs, statistical testing is unlikely to find it.

In a related study, Daran and Thevenod-Fosse [8] used a large number of test cases against twenty-four mutants and twelve real faults to determine if mutation testing is representative of real world faults. They generated a random test set from a uniform distribution, a structural coverage test set, and functional test sets of two types—from a finite state machine and from STATEMATE. They then applied all of these test sets to an implementation that contained a known fault. From the test cases that detected the real fault, they created another test set to apply to the mutants, using one test case from each test set category. They discovered that mutations can be as effective as real faults for determining the efficiency of a test suite. We use this discovery as the basis for our use of mutant specifications as systems under test. They also determined that random testing can be as effective as structural testing at detecting faults. Our case study takes this finding a step further by testing specifications of greater complexity than used in these studies.

In [10, 36, 35, 37], Ntafos, *et al.*, present the results of several testing experiments, pitting random testing against partition and required elements testing of implementations. *Partition testing* is any testing that selects at least one test case value from each subset of the input domain; the test case can be selected randomly from within the subdomain. *Required elements testing* selects an input value that follows a path through the model under test, from a variable definition through its subsequent references; this is the same as definition-use testing in [28]. In the first experiment [10], random testing was pitted against partition testing using a domain divided into 25 subdomains. This study used simulation to identify the fault-finding effectiveness of the partition testing strategy. The random testing strategy generated and executed test cases for three FORTRAN programs (SIN with three faults, SORT with 1 fault, and BIN with 1 fault). The results of these tests, in Table 5.4, show that random

testing performed better than partition testing in all cases except one. Column 3, # Test Sets, indicates how many test sets were generated for the program being tested; Column 4, Number Found, identifies how many of the test sets found the faults; and Column 5, Percent Found, presents the percentage of test sets that found the fault.

Table 5.4: Random Testing Results - Ntafos #1.

Testing Strategy	Program Tested	# Test Sets	Number Found	Percent Found
Partition	Simulated	50	14	28
Statistical	Sin1	50	11	22
	Sin2	50	24	48
	Sin3	50	45	90
	Sort	24	21	88
	Bin	50	18	36

In this test, two additional factors were discovered, (1) random testing was able to find what the authors called "relatively subtle" faults and (2) the randomly generated test sets also provided near total branch coverage.

The second experiment by Ntafos, *et al.* [35] pits random testing against a branch coverage test suite and a required elements test suite. Fourteen FORTRAN programs were tested, presenting an average of 468 mutants for each program to be killed. For branch and required elements testing, test cases were selected to satisfy the testing strategy and were the test sets were minimal test sets. For random testing, test cases were applied in groups of five until two consecutive groups failed to detect additional faults. Required elements testing fared best, followed closely by random testing and branch testing. Table 5.5 summarizes the results. Random testing outperformed branch testing in percentage of faults found but, on average, required nearly an order of magnitude more test cases (35 vs. 3.8). Required elements testing outperformed both random and branch testing, using a test case average between the two.

Even though required elements testing fared better, it required a directed graph of

Table 5.5: Testing Results - Ntafos #2.

Testing Strategy	Average # Test Sets	Percent Found
Random	35.0	93.65
Branch	3.8	91.60
Req. Elements	11.3	96.30

the system under test's behavior, which suffers from the same state space explosion as statistical testing when the models become more complex. Ntafos also notes that required elements testing uses more time to generate the test suite, then suggests that random testing is likely to be more cost-effective than required elements testing for larger systems.

In the final testing experiment conducted by Ntafos, *et al.* [36], proportional partition testing is compared to uniform partition testing. Proportional and uniform partition testing both choose test data randomly from the input subdomains, the difference being that uniform partition testing selects them equally from each of the subdomains, where proportional uses a probability figure on each subdomain to determine how many inputs are selected from each subdomain. Each test was performed 1000 times, with the results presented as better-equal-worse performance for the two strategies, based on the number of faults found. In Table 5.6, when 50 test cases are applied, partition testing outperforms random testing 5.5 to 1 and holds a 22.5 percent fault-finding advantage overall. When the number of test cases is raised to 2000, partition testing performs better overall, yet random testing still outperforms partition testing in 35 of the 1000 test sets and the fault-finding advantage of partition testing over random testing is reduced to 0.059 percent. In over half of the 2000 test sets, there is no difference in the fault-finding capability. This shows that random testing is a viable option if the generation of test cases is as fast or faster than that of a partition testing strategy.

Table 5.6: Testing Results - Ntafos #3.

Test Set Size	Comparison			Proportional Advantage
	Uniform	Equal	Proportional	
50	154	0	846	22.5
100	77	0	923	9.0
300	54	2	944	1.6
500	53	33	914	0.72
1000	45	262	693	0.22
1500	39	459	502	0.098
2000	35	581	384	0.059

Ntafos concludes that partition testing is more cost-effective if one test case is generated per subdomain. Additional test cases generated by a random testing strategy can even up the advantage and even swing the advantage in favor of random testing. The biggest drawback to the partition testing framework is the reliance on expert opinion to create the partitions. Since the most effective partitioning criteria can change dramatically between systems, it is nearly impossible to reuse partitioning criteria between systems under test. These criteria must be developed individually for each system under test.

5.5.2 Analytical studies

The other category of studies is analytical, where the test strategies or test suites are compared without actual testing. In a representative set of studies [13, 68, 69, 70], analysis is used to measure the relative strengths of random and branch testing criteria. They provide two relations, *BETTER* and *PROBBETTER*, which describe the effectiveness of two testing criteria relative to each other. These relations were developed to replace the SUBSUMES relation [48], which was shown to have little value when measuring effectiveness, as it did not indicate whether one strategy found more faults than another. A criterion C_1 SUBSUMES a criterion C_2 if, when C_1 is satis-

fied, C_2 is also satisfied. This is important, as it identifies criteria that may lead to a smaller test set. Unfortunately, the fact that one criterion subsumes another, does not indicate whether it finds more faults, as proven in [16]. The relation BETTER indicates that one strategy is more effective at finding faults than another. BETTER is defined as follows [70]: *Criterion C_1 is BETTER than criterion C_2, provided for every program P and specification S, any failure-causing input required by C_2 is also required by C_1.* This is formally different than the SUBSUMES relation, but often is the same in practice. Therefore, the same problem with misleading test cases can also occur here. Because of the possibility of misleading information from these two relations, [70] goes on to define a new relation, *PROBBETTER*. C_2 is PROBBETTER than C_1 indicates that a test set selected randomly from those satisfying criterion C_2 will more likely detect a fault than one randomly selected from a test set satisfying criterion C_1. The conclusion of these experiments is that PROBBETTER is a more valuable relation than BETTER and SUBSUMES and should be used when comparing the relative effectiveness of various testing strategies. We do not provide such a comparison in our testing but plan to explore these relationships in the future.

These testing experiments and studies discussed above provide useful insight and broad guidelines for our experiments, even though none of them is along the exact lines of what we aim to investigate. We use a specification as the basis for our testing, much as Offutt, Chevalley, and Thevenod-Fosse, but we use the specification as the basis of the operational profile as well as the oracle for test data generation. To our knowledge, we are the first to add input data probabilities to a specification with parallel constructs. With this background into related work and the testing framework we have created, we can now discuss the setup, execution, and results of our experiments.

Chapter 6

Experimental Results

As evidenced in the previous chapter, our experiments are designed as two sets of evaluations, a reflexive evaluation of statistical testing and a comparative evaluation of statistical and structural testing strategies. First, it is necessary to know how statistical testing can be used in the most effective, efficient manner. Secondly, we are interested in how it performs relative to another testing strategy, testing using a test suite generated to provide transition decision coverage.

6.1 Short vs. Long Test Cases

In this first set of experiments, we set out to determine what is an effective, efficient way to statistically test specifications. Based on our hypotheses presented earlier, we use differing configurations of test suites to check the fault-finding capability and coverage capability of each configuration.

One of the first tasks of our experiments is the definition of long and short test cases. To our knowledge, there is no definition for these in the literature and no one has performed these kinds of experiments with short and long test cases. As noted in the related works section, most experiments start with round numbers of steps, like 100 or 1000, and choose the input size based on multiples of this number (e.g., 1000, 2000,,10000). For our experiments, we define long, medium, and short with respect to our oracle model, the FGS. Prior studies at the University of Minnesota [20] have

determined that all states in the FGS05 model can be reached within ten steps; therefore, a test case with fewer than ten steps could be considered short. We arbitrarily picked four steps for our short test case length, as it is expected to be short enough to have some effect on the coverage and fault-finding capabilities of statistical testing.

For medium and long test cases, we chose to run a pilot study to see how coverage is affected as the test cases get longer. We doubled the size of the test case for each iteration of the study and measured the state, transition, and transition decision coverage for each iteration. We used values of 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4000, and 8000. The last two values were the results from earlier statistical testing experiments and required a significant amount of time to run, so we used those results rather than rerunning the tests at 4096 and 8192, as would be the true doublings. Since random generation of inputs could inadvertently generate one test case that performs extremely well for one test suite and one that performs extremely poorly for another, we ran each test suite generation and execution three times and averaged the results. The results of the coverage measurement for these tests are shown in Table 6.1.

By plotting these results in a graph, as shown in Figure 6.1, it is clear that the three coverage measures all increase rapidly as the test cases get longer but slow their increase dramatically at higher levels. The test set of size 0 provides the coverage measure in the initial state, to help visualize the growth of the coverage. We initialize the model and measure the state coverage immediately; it is not surprising that we have 0 for both transition and transition decision coverage in this case, as no transitions have occurred. The plotted coverage measure lines are nearly flat by the time we reach 8000 inputs, so we assumed little additional growth from this point on and stopped testing larger sizes. From the graph, we noted that the state coverage line's "knee," the point where the slope of the line is one, comes around sixty-four inputs,

Table 6.1: Long vs. Short Coverage Comparison.

TC Size	Coverage (%)			Test Time
	State	Trans	TD	
0	28.66	0.00	0.00	0
4	57.92	44.57	58.34	2
8	68.29	54.02	64.63	2
16	75.40	62.21	70.06	2
32	85.77	70.54	75.19	4
64	94.10	77.13	78.76	5
128	96.95	83.53	82.46	7
256	98.37	87.21	84.50	11
512	98.78	94.77	88.37	18
1024	98.78	93.41	87.69	32
2048	98.78	95.74	88.85	65
4000	98.78	95.34	88.66	120
8000	98.78	98.26	90.12	308

so a given increase in coverage requires an ever-increasing amount of test cases from this point onward. Since state coverage maximizes at 512 inputs and the transition and transition decision "knees" come around 512 inputs, for our experiments we chose to define medium as 64 inputs and long as 512 inputs. These numbers could easily be chosen at other points, but for our purposes, they are sufficient.

With these definitions of short, medium, and long, we then crafted the test suites. We hold the total number of input messages constant and vary the number of steps and test cases to build the test suites. In order to ensure that we exercise a realistic amount of the system under test, one test case is not sufficient. More than one would improve the chances of different transitions being taken from the initial state, thereby covering more of the system under test. We chose four test cases as the minimum number to generate for our experiments. By generating four long test cases of 512 steps each, we create and apply 2048 total inputs to the system under test. To provide equivalent numbers of inputs, the medium test suite requires 32 test cases of 64 steps

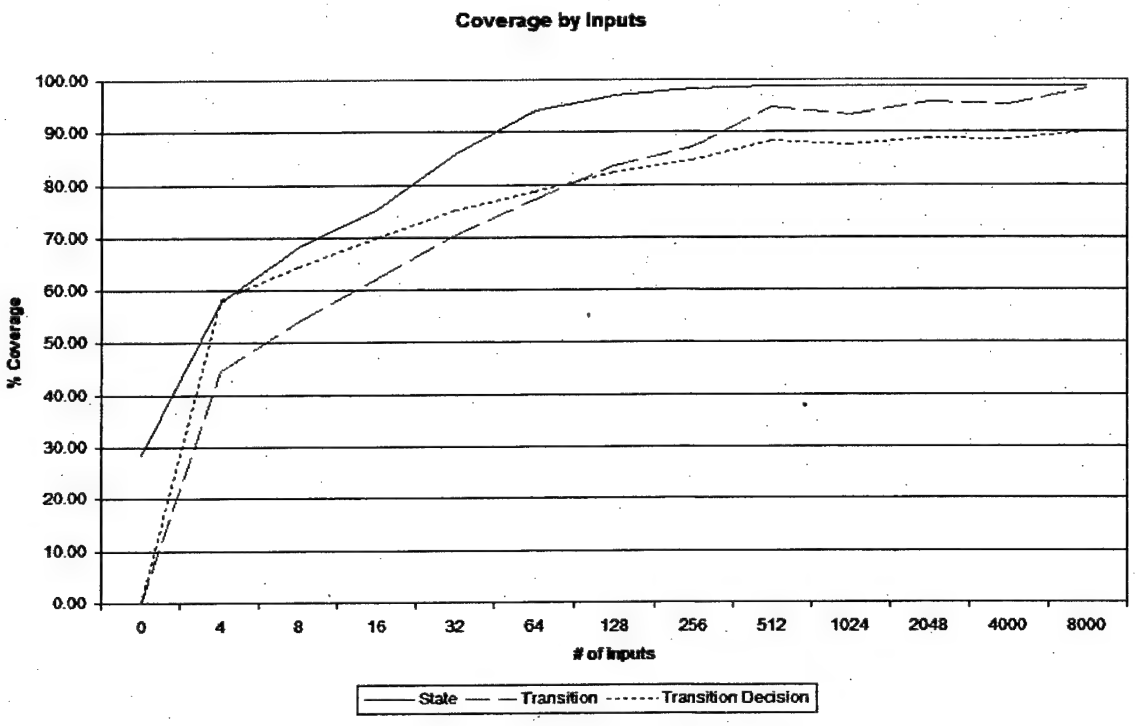

Figure 6.1: Long vs. Short Graph.

each and the short test suite requires 512 test cases of 4 steps each. With these test suite configurations, we outline our long versus short experiments, present the results, and make some conclusions regarding the ability of short, medium and long test cases for providing system coverage and for finding faults.

6.1.1 Coverage

In the first experiment, we use our coverage analyzer in the NIMBUS simulator to apply the long, medium, and short test suites to the oracle specification and measure the level of state, transition, and transition decision coverage provided by each.

Hypothesis 1: *Given an equivalent number of input messages, uniform statistical testing using many short test cases provides greater state, transition, and transition*

decision coverage than uniform statistical testing using fewer long test cases.

Results

We expected state coverage to be moot for our experiments, since the uniform distribution makes inputs equally likely and the state values should all be encountered very quickly. Only very short test sequences were expected to provide less than complete state coverage. In our experiment, we discovered that complete state coverage eluded us for random testing, as shown in Table 6.2. Further investigation of the model revealed two Boolean state variables that have no transitions attached to them, thereby causing each to hold its initial value throughout execution. Our state coverage analysis application counts the possible values from the type definitions, not from the actual reachable values; therefore, the highest reachable state coverage is 98.78 percent, as attained by both the medium and long test suites. The short test suite comes very close but misses one state value the medium and long test suites reach.

Table 6.2: Long vs. Short Coverage Comparison.

Testing Strategy	(Steps x TCs)	Coverage			Total Time
		State	Transition	Trans. Dec.	
Short	(4 x 512)	98.37	94.77	88.28	190
Medium	(64 x 32)	98.78	98.64	90.31	73
Long	(512 x 4)	98.78	95.74	88.85	62

For transition and transition decision coverage, we were at first surprised at the results of the coverage comparison. We expected the medium test suite to cover more of the system than the short test suite, which it did, but we expected the long test suite to cover more than both of them. This was not the case. Since the long test cases are longer than the medium test cases, depth is not likely to be the cause of the lower coverage. Rather, it is more likely that choices early in the test may preclude certain

portions of the model from being reached. More test cases increase the likelihood of alternate choices being made early in the execution of the test. Therefore, it is necessary to create test suites that provide enough depth *and* breadth of coverage in a statistical test suite.

6.1.2 Fault-finding

In addition to the coverage comparison, we used the generated test suites to compare the fault-finding capability of short, medium, and long test cases. These were run against the one hundred mutant specifications to determine how many faults each could find.

Hypothesis 2: *Given an equivalent number of input messages, uniform statistical testing using many short test cases finds more faults than uniform statistical testing using fewer long test cases.*

Results

From the results of the fault-finding capability tests, in Table 6.3, we found that the fault-finding ability of the different test suite configurations was not significantly different. In fact, the medium and long configurations had equal numbers of faults found for the variable reference and condition insertion fault types, and short was only slightly less effective. Even for the condition negation and condition removal faults, the numbers are separated by one and two faults found, respectively. The similar numbers indicate that the length of the test case is not as important as the number of messages in the entire test suite. Also, of the 65 faults found by the medium test suite, only three were not in common with the long test suite and all of the ones found by the short test suite were found by the medium test suite.

Table 6.3: Long/Medium/Short Fault-finding Comparison.

Fault Type	Number Seeded	Faults Found		
		Long	Med	Short
VRF	25	21	21	20
CIF	25	4	4	1
CNF	25	24	25	25
CRF	25	14	15	13
Total	100	63	65	59

The few extra faults that were missed by the short test suite are most likely due to the depth of the system state space being more than the four steps of the test case. Faults in portions of the state space that are deeper than four steps are beyond the reach of the short test suite. On the other hand, the faults missed by the long test suite are most likely due to the small number of test cases not providing the breadth of coverage of the model. Therefore, from the straight count of faults found, the length of the test cases does not appear to be a major factor in fault-finding, provided the test cases are long enough to cover the depth of the state space and there are enough test cases to cover the breadth of the state space.

For these two hypotheses, either the medium or long test suite configuration would be preferable, as they provide nearly the same level of coverage and fault-finding. Since these are comparable, the deciding factor would be to determine which can be generated and executed in the least amount of time. As we performed these experiments, we recorded the time required for the generation and execution of the above test suites. From this information, shown in Table 6.4, it is evident that the longer test suite takes less time for generation and execution, making it the best choice for testing. Again, we noted that the number of test cases would have to be increased to provide breadth and depth coverage of the system under test.

There is a time penalty incurred for resetting the simulator to execute each test case. The simulator is reset for each test case and has to connect to the appropriate

Table 6.4: Long/Medium/Short Time Comparison.

TC Length	Number of Steps	Test Cases	Time(seconds)		
			Gen	Test	Total
Long	512	4	24	38	62
Medium	64	32	25	48	73
Short	4	512	31	159	190

input message files, both of which require additional time. The simulator also records the state of the system under test at the end of each step and a new system log must be created for each test case. These file system actions are quite costly with respect to the simulator's internal actions, evident in the fact that the short test suite took more than twice as long as the medium test suite and almost three times as long as the long test suite.

From these results, it is apparent that a test suite comprised of fewer long test cases is the best choice for a statistical testing strategy, at least for the case example discussed here. The state coverage measures are equal for the medium and long and better than the short, the fault-finding capability for the medium and long test suites is comparable and significantly better than for the short test suite, and the long test suite has somewhat faster performance than either the medium or short test suites. We should choose a test case strategy that uses longer test cases but also generates enough test cases to provide better model coverage. This strategy is the one we use to compare with structural testing.

6.2 Statistical Testing vs. Structural Testing

As noted in the experimental setup chapter, we use the results of a structural testing strategy experiment for comparison to statistical testing. The baseline figures for structural testing, to be used to determine statistical testing effort, are shown in

Table 6.5. The coverage figures will be discussed fully in Sections 6.2.1 and 6.2.2. Two test suites were generated, one to provide state coverage and one to provide transition decision coverage, as defined in Section 5.3.6. We converted the test suites, in ITR format, into input messages and applied them to the oracle specification to determine each suite's total testing time. We chose to use the transition decision test suite as the baseline for total testing time, as it provides coverage comparable to the uniform statistical testing strategy.

Table 6.5: Structural Testing Baseline Figures.

Coverage Measure	# TCs	Gen Time	Test Time	Total Time	Coverage		
					State	Trans	TD
State	115	82	46	128	98.78	77.91	78.20
TD	313	194	58	252	98.78	100.0	90.99

Based on these results, we could choose either the medium or the long test case generation strategy, as both fall well within the time used by the transition decision test suite. If we use the medium test case generation strategy, we can run nearly three and a half times as many test cases in the time it takes to run the transition decision test suite; the long test case generation strategy can run four times as many test cases in this same time period. From the earlier discussion, we showed that the long test case generation strategy had better depth for the faults it found but more test cases were needed to provide higher coverage of the system. Also, the medium test case generation strategy provided better coverage and fault-finding, since it had more test cases, but it incurred a greater penalty because of those test cases. Therefore, we need to make the test cases longer in the medium test case generation strategy or include more test cases in the long test case generation strategy. In the medium test case generation strategy, to increase the test case length by a factor of approximately 3.45 (252/73), it results in 32 test cases of 220 steps or 7040 total inputs. On the other hand, running the long test case generation strategy four times would result in

a total of 8192 inputs in just under the 252 seconds. Since we know that the number of test inputs is a primary factor in fault-detection and coverage measures, we chose the longer test case generation strategy, expecting that the additional test cases will make up the difference in the coverage and fault-finding measures.

6.2.1 Coverage

The first experiment of this section is a comparison of the coverage provided by the statistical and structural testing strategies. The uniform statistical testing test suite described above was generated and executed in the same manner as in the earlier experiments of hypotheses 1 and 2.

Hypothesis 3: *Given equal effort in generating and applying test cases, uniform statistical testing can provide equivalent levels of state, transition, and transition decision coverage, compared to structural coverage testing designed to provide transition decision coverage.*

Results

Our results, in Table 6.6, show how statistical and structural testing compared in our experiment. We performed structural testing using both a state coverage test suite and a transition decision coverage test suite.

Table 6.6: Testing Strategy Coverage.

Testing Strategy →	Statistical	Structural	
Coverage Metric ↓		TD	State
State	98.78	98.78	98.78
Transition	98.84	100.00	77.91
Transition Decision	90.41	90.99	78.20

As noted in Section 6.1.1, the highest state coverage that can be reached with the

FGS05 model is 98.78 percent, as two Boolean state variables have no transitions and cannot take on their second value. Here we can see that all three test suites provide equal levels of state coverage.

For transition and transition decision coverage, we did not expect the state coverage package to do very well and we were not surprised at the results. Simply covering all state values is not adequate for providing transition coverage of a system. The transition decision test suite performed nearly as expected and produced 100 percent transition coverage and 90.99 percent transition decision coverage. Further investigation of the FGS model showed that there are thirty-one transitions whose conditions are of the form `EQUALS new_state IF TRUE`. These transitions can never be set to false, so they can never be included in the transition decision count. When these are accounted for, the maximum attainable transition decision coverage is 90.99 percent, which is exactly what the transition decision test suite provided. In that same light, the uniform statistical test suite provided nearly the same level of both transition and transition decision coverage as the transition decision test suite. Therefore, our experiments support the hypothesis that uniform statistical testing seems to provide comparable levels of state, transition, and transition decision coverage.

6.2.2 Fault-finding

Since the coverage measures are nearly equivalent for statistical and structural testing, it becomes even more interesting to compare the fault-finding capabilities of the two test suites. In this experiment, we execute the statistical test suite and the transition decision coverage test suite against the mutant specifications and compare the results to provide some insight into the relative efficiency of the two testing strategies.

Hypothesis 4: *Given equal effort in generating and applying test cases, uniform statistical testing can detect equivalent numbers and types of faults, compared to structural coverage testing designed to provide transition decision coverage.*

Results

The structural coverage strategy is designed to reach every transition of the model and exercise both true and false values for each transition. This strategy would be expected to do well in detecting faults in the specifications. We did not find this to be true in our experiments. Figure 6.2 and Table 6.7 present the results of our testing.

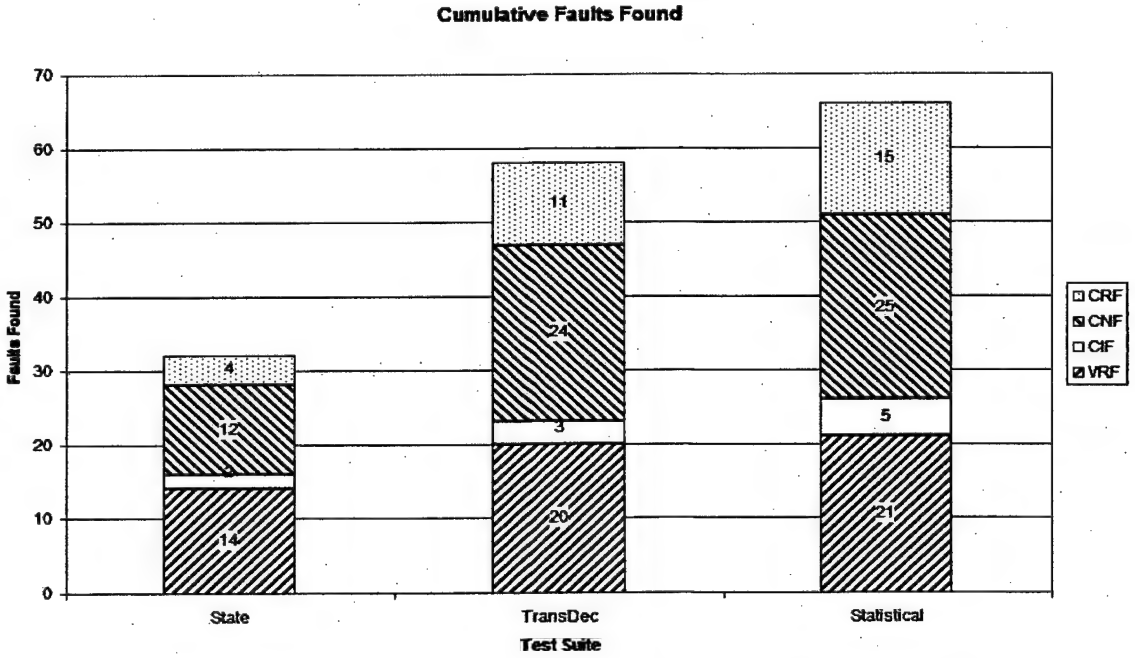

Figure 6.2: Structural vs. Statistical Fault-finding Comparison.

It is evident that, as this experiment is constructed, uniform statistical testing not only could find as many faults as structural testing, it actually found more of them. In fact, in our experiment, structural testing could find only one fault that uniform

Table 6.7: Fault-finding Capability Comparison.

Fault Type	# Seeded	Statistical	Structural
Variable Replacement	25	21	20
Condition Insertion	25	5	3
Condition Negation	25	25	24
Condition Removal	25	15	11
Total	100	66	58

statistical testing could not find; uniform statistical testing found eight that structural could not. A complete listing of our fault-finding results is shown in Table 6.8 and includes all structural and statistical configurations we tested.

6.3 Case Study Results Discussion

Our initial reaction to these results was to question our implementation of the test case generation—we simply suspected that we erroneously generated tests that did not provide the desired coverage. A close examination of the test suites and specification coverage measures confirmed that the test suites provided the desired coverage—they just did not find many flaws. We found the problems were related to (1) a model structure that 'cheats' the coverage criteria and (2) coverage criteria that are inadequate with respect to the semantics of our specification language, as well as other common languages. Below we elaborate on these two issues.

6.3.1 Model Structure

As mentioned above, the test case length was universally very short and, seemingly, quite poor at revealing faults. To explain this phenomenon, let us take a closer look at the flight guidance system used as a case example.

The FGS is part of a larger flight control system (FCS) that, among many other components, contains two FGSs—a left FGS and a right FGS (one for the pilot and

one for the co-pilot). In most situations, only one FGS is actually flying the aircraft (it is the active FGS) and the other FGS (the inactive FGS) behaves as a hot spare, receiving all of its state information from the active FGS. In short, there is an interface between the FGSs through which they can control each other and, if it is the active FGS, command the other FGS into arbitrary state configurations.

Most test cases that the model checker found took advantage of this particular feature of the FGS model—the test cases made the FGS under test the inactive FGS and simply used the other FGS interface to drive the model to the desired state (or take the desired transition, or make the desired condition true or false). For example, when the FGS is active, there are some rather complex rules for when to enter the ROLL mode. When the FGS is inactive, on the other hand, all that is required to enter the ROLL mode is to command it there with an input variable. Thus, is is possible for test cases to achieve coverage by commanding the FGS to go to states and take transitions—they do not exercise the actual mode logic of the FGS. Naturally, such test cases will not reveal any faults seeded in the mode logic of the FGS. Unfortunately, this is exactly the test case that the bounded model checker is likely to find—it is most often the simplest possible way of achieving the test objective.

To solve this problem, we can simply prohibit messages on the FGS-to-FGS interface and in that way force the model checker to find test cases that actually use the actual mode logic. For example, we can add an invariant to the model checker prohibiting the FGS from being inactive.

```
INVAR (Is_This_Side_Active = 1)
```

This is technically quite an easy thing to do, but it requires intimate knowledge of the existence of the FGS-to-FGS interface and the knowledge that this interface can be "abused" by the test case generation automation to achieve its objectives. From this

work we have come to the conclusion that state and transition coverage are clearly inadequate in this domain—more elaborate coverage is necessary. We have some hope for various condition based coverage, for example, modified decision and condition coverage (MC/DC) [7], but our experiences with simple decision coverage (discussed next) raises some issues with the adoption of condition-based coverage criteria in the specification domain.

6.3.2 Inadequate Coverage Criteria

To illustrate the problems with condition-based coverage criteria, we consider this small code fragment:

```
TRANSITION state1 TO state2 IF
  TABLE
    Var1 : T * ;
    Var2 : * T ;
  END TABLE
TRANSITION state2 TO state3 IF Var3
```

Two test inputs, `Var1 = TRUE` and `Var3 = TRUE`, would achieve transition coverage of the first transition, as the transition evaluates to true on the first input and false on the second. However, there is no requirement that all variables and conditions in the transition are even evaluated. For example, if the language has lazy-evaluation (as in NIMBUS) and a fault is located in the second column of the first transition (T changed to F), the second column does not need to be evaluated in order to get transition coverage of this transition and the fault will go undetected.

This is exactly the problem we face with the condition-based coverage criteria such as decision coverage and MC/DC coverage—if the decision of interest is masked out or never evaluated, the test is not particularly useful. To our knowledge, the condition-based coverage criteria required in practice (for example, in certification to DO-178B [55]) and used in previous studies do not require us to take usage information

into account. As can be seen in the data in Table 6.7, transition coverage did not reveal as many faults as we expected, largely because the tests generated satisfied the 'letter' of the criterion—the inputs make the transition true and false—but not the 'spirit' of the criterion—the transition is never fully evaluated. To solve this problem, the coverage criteria must be modified to take data flow into account—the criteria must assure that somehow the individual decisions are invoked. The unanswered question is how we can modify the requirements on the test suite to require that the entire transition decision component is exercised in some way by the test suite.

Summary of Experiments

Although structural testing provided slightly better coverage, statistical testing performed better than structural testing in our fault-finding experiments. This is not to say that it will perform better in all experiments, this is only one experiment covering one particular system. Our goal was twofold–to evaluate using a specification model as an operational profile to generate test suites and to evaluate the resulting test suites against a structural testing test suite. We accomplished both of these satisfactorily. As for our framework, then, we can make the following assertions about long versus short test case configurations:

1. Longer test cases are better for statistical testing, as significant time is spent in resetting the test case generator and the testing framework for each test case. However, one extremely long test case is not a realistic configuration and may take transitions in the early portions of the state exploration that may preclude exercising the entire reachable state space. Therefore, the long test cases must be numerous enough in the test suite to provide breadth of coverage of the system under test as well as adequate depth of coverage.

2. Many short test cases do no provide as high a level of coverage as fewer long test cases. Short test cases will miss the deeper depths of the state space.

3. Many short test cases are not better at finding faults as fewer long test cases. Again, short test cases will miss faults that are at deeper depths of the state space.

For the statistical versus structural testing strategies, we confirmed the belief that structural testing will outperform statistical testing in coverage measures. The best we can hope for is for statistical testing to reach comparable levels of coverage, which it did in the same amount of time. With the addition of a true operational profile, it is expected that statistical testing will take even longer to reach the highest levels of coverage, due to input combinations with low probabilities.

For fault-finding, the opposite is true, statistical testing performed significantly better than the transition decision structural testing strategy. This is due to the presence of multiple conditions for each transition and the structural test suite generator only needing to satisfy one condition. Also, transition decision coverage likely could not find certain faults due to a "back door" input message that allows the structural test suite generator to bypass some of the mode logic of the FGS in order to find the shortest test case. The primary lesson to learn from this experiment is that generating a test suite to provide transition decision coverage is not the best strategy for finding faults in specifications like the FGS. With the multiple conditions in transitions, transition decision coverage does not equate to testing all parts of the transition condition logic. This is where strategies like clause-wise guard coverage [50], MC/DC [7], and full predicate coverage [41] testing come into play.

6.4 Caveats

There are some limitations and assumptions on the testing that could affect the outcomes of future tests.

Speed of Testing Framework: A major factor in this test strategy comparison is the speed of the test framework. The NIMBUS simulator was designed for flexibility and not optimized for individual functions like test case generation. Therefore, it is not particularly quick for any specific function. On the other side, the model checker for structural test case generation is a very fast application. A faster statistical test suite generator would allow for a greater number of tests to be created and run in the same amount of time used by the structural test suite generator. Secondly, we also use the NIMBUS simulator for executing the test cases and speed is again a limiting factor. By increasing the speed of test execution, we could generate and execute significantly more statistical test cases in the given time.

Random vs. Statistical Testing: In our experiments, we used a uniform operational profile, which results in the equivalent of random testing with multiple inputs. This results in two issues of concern, the number of test cases required and the perceived reliability of the system under test. Using a uniform distribution requires fewer test cases to provide coverage of the state space, compared to a more realistic operational profile, as the presence of low probabilities on some inputs can require significantly more test cases to ensure those inputs are generated.

The counterpoint to this issue is the reliability of the system being tested. By using a uniform distribution for choosing input data, we are likely overtesting little-used portions of the system and undertesting heavily-used portions, resulting in a product that could still be viewed as having lower reliability. A true operational profile will

direct more testing to the heavily-used portions, resulting in higher reliability of the system. The research of this dissertation sets the stage for future experiments with operational profiles, to determine how changing probabilities will affect the coverage and fault-finding capability of statistical testing.

Fault Seeding: As noted in Section 5.3, we were not concerned with eliminating duplicate faults or mutant specifications that are semantically equivalent to the source model. The presence of either does not change the outcome of our testing, as the first should be found or not found each time it is present and the second should not be found by either testing strategy. Therefore, even though random testing detected two-thirds of the seeded faults, its actual performance could be much higher if the remaining faulty specifications were actually semantically equivalent specifications.

Table 6.8: Raw Fault-finding Data.

	Variable Replacement Fault																								
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	Found
State			X	X	X			X	X	X	X	X	X			X					X		X	X	X	14
TransDec		X	X	X	X	X		X	X	X	X	X	X		X	X	X		X	X	X		X	X	X	20
Statistical		X	X	X	X	X	X	X	X	X	X	X	X		X	X			X	X	X	X	X	X	X	21
Short		X	X		X	X	X	X	X	X	X	X	X		X	X			X	X	X	X	X	X	X	20
Medium	X	X	X		X	X	X	X	X	X	X	X	X		X	X			X	X	X	X	X	X	X	21
Long		X	X	X	X	X	X	X	X	X	X	X	X		X	X			X	X	X	X	X	X	X	21

	Condition Insertion Fault																									
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	Found
State										X						X										2
TransDec										X						X		X								3
Compare										X	X			X		X		X								5
Short																X										1
Medium										X	X					X		X								4
Long										X	X					X		X								4

	Condition Negation Fault																									
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	Found
State		X	X		X		X	X	X				X				X			X	X	X			X	12
TransDec	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X	X	X	X	X	X	X	24
Compare	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	25
Short	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	25
Medium	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	25
Long	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X	24

	Condition Removal Fault																									
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	Found
State		X																X	X						X	4
TransDec		X	X					X		X		X	X				X	X	X		X				X	11
Compare	X	X	X		X			X		X		X	X			X	X	X	X		X			X	X	15
Short	X	X	X		X			X				X	X				X	X	X		X			X	X	13
Medium	X	X	X		X			X		X		X	X			X	X	X	X		X			X	X	15
Long	X	X	X		X			X		X		X	X			X	X		X		X			X	X	14

Chapter 7

Future Research and Summary

7.1 Future Direction of Research

In this dissertation, we made the following contributions:

- We demonstrated the use of state-based specification models as the basis of an operational profile for statistical testing. This allows us to model systems of greater complexity and fidelity than what is currently used in statistical testing. We identified conditional probabilities for input data and extended the specification language RSML^{-e} to model these conditional probabilities.

- We presented a statistical testing framework that successfully generates test cases according to an operational profile created from a state-based requirements specification and executes these test cases on a system under test.

- We showed that statistical testing performs well compared with test suites generated to meet common structural coverage criteria. The state, transition, and transition decision coverage measures of the test suites generated by our statistical testing framework were comparable to the coverage measures of structural coverage test suites. Further, the fault finding capability of the statistically generated test suites exceeded that of the test suites generated to meet structural coverage criteria. This finding is instrumental in identifying the need for better definition of coverage measures, especially for use in generating test suites.

With the new statistical testing capability provided through these contributions, there are several areas of research that can further extend statistical testing and deserve investigation in the future:

Integer and real type variables: The infinite domains of these variable types pose problems in the definition of the requirements specification used as the source model and in the statistical component of the operation profile. Both of these areas need to be addressed in order to improve the fidelity of the operational profile.

The requirements specification used as source model: Types with infinite domains cause state space explosion; however, they may be necessary to accurately define the specification. Therefore, they need to be addressed to keep the state space manageable. By implementing common abstraction techniques, these variables can be reduced to equivalence classes, allowing for a significantly smaller state space, yet providing necessary detail for testability. Our project addresses Boolean and enumerated types but should be adapted to include integer and real types.

Infinite types in the operational profile: Our FGS model has only finite domain types for input message fields (Booleans and enumerated). These are straightforward to model; infinite types are not as easy to model explicitly. One potential solution to this problem is to allow a probability function or expression to be used (for example, that allows an altitude input be picked randomly from within a range, such as plus or minus one percent).

Observability of test case result: It is difficult to determine the system state at a particular time of execution in most operating environments, especially if there

is no output or other observable information. This is an important concept in our research, as the outcome of a successful test case may be only a change of state, with no corresponding output. NIMBUS allows us access to all state variables, which makes it straightforward to compare the current state of the system to the expected state. To accurately record the test case execution in other testing frameworks will require either instrumentation of the executable code to output the system state or some form of application or daemon that can monitor the system state.

Collection of profile data: Reliability estimation and business decision-making would benefit from the construction of highly accurate operational profiles. We now have the capability to model more complex systems but we need to collect actual usage data to crete accurate operational profiles. This could be accomplished by instrumentation of code to output the system state along with inputs and outputs or the creation of a daemon that could watch the process and record inputs, outputs, and system state while a system is in operation.

Statistical analysis: Statistical analysis of systems is performed on a transition matrix, which is required to be a square matrix, n x n, where n is the number of states in the state space. This becomes the breaking point as systems become more complex. Also, the matrix is often sparsely populated, leading to a large amount of wasted space. New techniques are needed for storing the matrix or new methods are needed for computing the various quality metrics for a model.

7.2 Summary

It is evident through this research that we can use a state-based requirements specification as the basis for an operational profile for statistical testing of software. We

can capture the structural model in the specification, we can identify where to place the probabilities for statistical testing, and we can extend the modeling language to include these probabilities.

With these probabilities identified and modeled, we can generate test suites on a statistical basis, testing a system as it is likely to be used in actual operation. The generation of these tests is quick, allowing for many test cases to be generated and applied in a short period of time.

Bibliography

[1] Kaushal Agrawal and James A. Whittaker. Experiences in applying statistical testing to a real-time, embedded software system. *Proceedings of the Pacific Northwest Software Quality Conference*, pages 154–170, 1993.

[2] Paul E. Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of the Fourth IEEE International Symposium on High-Assurance Systems Engineering*. IEEE Computer Society, November 1999.

[3] Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, November 1998.

[4] Paul E. Black, V. Okun, and Y. Yesha. Mutation operators for specifications. *Proceedings of the Automated Software Engineering (ASE)*, pages 81–88, 2000.

[5] R. W. Butler and G. B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19(1), January 93.

[6] Philippe Chevalley and Pascale Thvenod-Fosse. Automated generation of statistical test cases from uml state diagrams. *25th Annual International Computer Software and Applications Conference*, pages 205–214, October 2001.

[7] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, September 1994.

[8] Murial Daran and Pascale Thvenod-Fosse. Software error analysis: a real case study involving real faults and mutations. In *Proceedings of the 1996 international symposium on Software testing and analysis*, pages 158–171. ACM Press, 1996.

[9] K. Doerner and W. J. Gutjahr. Representation and optimization of software usage models with non-Markovian state transitions. *Information and Software Technology*, 42(12):873–887, 2000.

[10] Joe W. Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th international conference on Software engineering*, pages 179–183, 1981.

[11] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking.* MIT Press, 1999.

[12] Pablo Fernandes, Brigitte Plateau, and William J. Stewart. Efficient descriptor-vector multiplications in stochastic automata networks. *Association for Computing Machinery*, 45(3), May 1998.

[13] Phyllis G. Frankl and Elaine J. Weyuker. Provable improvements on branch testing. *IEEE Transactions on Software Engineering*, 19(10):962–975, October 1993.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, 1995.

[15] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, November 1999.

[16] Dick Hamlet. Theoretical comparison of testing methods. In *Third Symposium on Software Testing, Analysis, and Verification*, pages 28–37. ACM SIGSOFT, December 1989.

[17] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[18] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[19] M. P.E. Heimdahl and N.G. Leveson. Completeness and Consistency Analysis of State-Based Requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.

[20] Mats P.E. Heimdahl, Sanjai Rayadurgam, Willem Visser, George Devaraj, and Jimin Gao. Auto-generating test sequences using model checkers: A case study. *3rd International Workshop on Formal Approaches to Testing of Software*, pages 44–62, October 2003.

[21] Mats P.E. Heimdahl, Jeffrey M. Thompson, and Barbara J. Czerny. Specification and analysis of intercomponent communication. *IEEE Computer*, pages 47–54, April 1998.

[22] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance, COMPASS 95*, 1995.

[23] C. L. Heitmeyer, B. L. Labaw, and D. Kiskis. Consistency checking of SCR-style requirements specifications. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, March 1995.

[24] Christophe Hirel. Reliability and performability modeling using SHARPE. *11th InternationalConference on TOOLS*, March 2000.

[25] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '02)*, Grenoble, France, April 2002.

[26] Erlangen-Twente Markov Chain Checker: User's Guide-Version 1.3. Technical report, Institute for Computer Science, University of Erlangen-Nurnberg, Germany, http://www7.informatik.uni-erlangen.de/etmcc, Feb 2001.

[27] Paul L. Jones, W. Thomas Swain, and Carmen J. Trammell. Engineering in software testing: Statistical testing based on a usage model applied to medical device development. *Biomedical Instrumentation and Technology*, pages 311–322, Jul 1999.

[28] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach.* CRC Press, Boca Raton, FL, 1995.

[29] Anjali Joshi, Steven P. Miller, and Mats P. E. Heimdahl. Mode confusion analysis of a flight guidance system using formal methods. *To appear in Digital Avionics Systems Conference*, October 2003.

[30] David P. Kelly and Rob S. Oshana. Improving software quality using statistical testing techniques. *Information and Software Technology*, 42(12):801–807, 2000.

[31] Ara Kouchakdjian and R. Fietkiewicz. Improving a product with usage-based testing. *Information and Software Technology*, 42(12):809–814, 2000.

[32] D. R. Kuhn. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering Methodology*, 8(4):411–424, October 1999.

[33] Marta Kwiatkowska. Prism: Probabilistic Symbolic Model Checker. Technical report, University of Birmingham, United Kingdom, http://www.cs.bham.ac.uk/ dxp/prism.

[34] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–706, September 1994.

[35] Simeon C. Ntafos. An evaluation of required element testing strategies. In *Proceedings of the 7th International Conference on Software Engineering*, pages 250–256, 1984.

[36] Simeon C. Ntafos. On random and partition testing. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 42–48. ACM Press, 1998.

[37] Simeon C. Ntafos. On comparisons of random, partition, and proportional partition testing. *IEEE Transactions on Software Engineering*, 27(10):949–960, October 2001. Similar to Ntafos98 and Ntafos84.

[38] University of Tennessee Software Quality Research Laboratory. Using the JUMBL. http://www.cs.utk.edu/sqrl/esp/jumbl4/jumbl_using.html, 2002.

[39] A. Jefferson Offutt and Ayour Abdurazik. Generating tests from UML specifications. *2nd International Conference on Unified Modeling Language*, pages 416–429, October 1999.

[40] A. Jefferson Offutt, Shaouing Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification, and Reliability*, 13(1):25–53, January/March 2003.

[41] A. Jefferson Offutt and Jie Pan. Detecting equivalent mutants and the feasible path problem. *11th Annual Conference on Computer Assurance*, pages 224–236, June 1996.

[42] A. Jefferson Offutt, Yiwie Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, October 1999.

[43] Alberto Pasquini, E. D. Agostine, and G. DiMarco. An input-domain based method to estimate software reliability. *IEEE Transactions on Reliability*, pages 95–105, March 1996.

[44] Jesse H. Poore. Introduction to the special issue on: Model-based statistical testing of software intensive systems. *Information and Software Technology*, 42(12):797–799, 2000.

[45] Jesse H. Poore, Gwen H. Walton, and James A. Whittaker. A constraint-based approach to the representation of software usage models. *Information and Software Technology*, 42(12):825–833, 2000.

[46] Stacy J. Prowell. TML: A description language for Markov chain usage models. *Information and Software Technology*, 42(12):835–844, 2000.

[47] Stacy J. Prowell, Carmen J. Trammell, Richard C. Linger, and Jesse H. Poore. *Cleanroom Software Engineering.* Addison-Wesley, Reading, Massachusetts, 1999.

[48] S. Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[49] Sanjai Rayadurgam. *Automatic Test-case Generation from Formal Models of Software.* PhD thesis, University of Minnesota, November 2003.

[50] Sanjai Rayadurgam and Mats Heimdahl. Automated test-sequence generation from formal requirements models. *Sixth International Symposium on High Assurance Systems Engineering*, pages 28–31, October 2001.

[51] Sanjai Rayadurgam and Mats P.E. Heimdahl. Coverage based test-case generation using model checkers. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*, pages 83–91. IEEE Computer Society, April 2001.

[52] Sanjai Rayadurgam and Mats P.E. Heimdahl. Coverage based test data generation using model checkers. Technical Report 01-005, Dept. of Computer Science and Engineering, University of Minnesota, Minneapolis, January 2001.

[53] Sanjai Rayadurgam and Mats P.E. Heimdahl. Test-Sequence Generation from Formal Requirement Models. In *Proceedings of the 6th IEEE International Symposium on High Assurance Systems Engineering (HASE 2001)*, Boca Raton, Florida, October 2001.

[54] Björn Regnell, Per Runeson, and Claes Wohlin. Towards integration of use case modelling and usage-based testing. *The Journal of Systems and Software*, 50(2):117–130, February 2000.

[55] RTCA. *Software Considerations In Airborne Systems and Equipment Certification.* RTCA, 1992.

[56] Rosaria S and Robinson H. Applying models in your testing process. *Information and Software Technology*, 42(12):815–824, 2000.

[57] Kirk Sayre and Jesse H. Poore. Partition testing with usage models. *Information and Software Technology*, 42(12):845–850, 2000.

[58] Kirk Sayre and Jesse H. Poore. Stopping criteria for statistical testing. *Information and Software Technology*, 42(12):851–857, 2000.

[59] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1992.

[60] William J. Stewart. MARCA: Markov chain analyzer. Technical report, North Carolina State University, http://www.cse.ncsu.edu/faculty/Wstewart /MARCA/marca.html.

[61] K. Tewary and Mary Jane Harrold. Fault modeling using the program dependence graph. *Proceedings of the Fifth International Symposium on Software Reliability Engineering*, pages 126–135, November 94.

[62] Pascale Thevenod-Fosse, Helene Waeselynck, and Y. Crouzet. An experimental study on software structural testing: Deterministic versus random input generation. *Twenty-First International Symposium on Fault-Tolerant Computing*, pages 410–417, June 1991.

[63] Michael G. Thomason and James A.Whittaker. Rare failure-state in a Markov chain model for software reliability. *10th International Symposium on Software Reliability Engineering (ISSRE)*, Nov 1999.

[64] Jeffrey M. Thompson, Mats P.E. Heimdahl, and Michael W. Whalen. The NIMBUS environment for specification of safety critical systems. Technical report, University of Minnesota, Apr 2001.

[65] Jeffrey Michael Thompson. *Structuring Formal State-Based Specifications for Reuse and the Development of Product Families.* PhD thesis, University of Minnesota, 2002.

[66] T. Tsuchiya and T. Kikuno. On fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engineering Methodology*, 11(1):58–62, January 2002.

[67] Gwen H. Walton and Jesse H. Poore. Measuring complexity and coverage of software specifications. *Information and Software Technology*, 42(12):859–872, 2000.

[68] Elaine J. Weyuker. Can we measure software testing effectiveness? *First International Software Metrics Symposium*, pages 100–107, May 1993.

[69] Elaine J. Weyuker and Bingchiang Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[70] Elaine J. Weyuker, Stewart N. Weiss, and Dick Hamlet. Comparison of program testing strategies. In *Proceedings of the symposium on Testing, analysis, and verification*, pages 1–10. ACM Press, 1991.

[71] Michael W. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.

[72] James A. Whittaker. Stochastic software testing. *Annals of Software Engineering*, 4:115–131, 1997.

[73] James A. Whittaker and Jesse H. Poore. Markov analysis of software specifications. *ACM Transactions on Software Engineering and Methodology*, 2(1):93–106, Jan 1993.

[74] James A. Whittaker and Michael G. Thomason. A Markov chain model for statistical software testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, Oct 1994.

[75] H. Zhu, P.A.V. Hall, and J.H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

Appendix A

RSML^{-e} Statistical Profile Model

The following is an example of the statistical component of our Requirements State Machine Language without Events (RSML^{-e}) operational profile:

```
INTERFACE_PROBABILITY
   DEFAULT :
      This_Input  [0.50];
      Other_Input [0.50];
END INTERFACE_PROBABILITY

MESSAGE_PROBABILITY
   This_Input :
      DEFAULT :
         AltPreRefChanged        : FALSE [0.50], TRUE [0.50];
         AltselCaptureCondMet    : FALSE [0.50], TRUE [0.50];
         AltselTargetAltChanged  : FALSE [0.50], TRUE [0.50];
         AltselTrackCondMet      : FALSE [0.50], TRUE [0.50];
         AltSwi                  : OFF [0.50], ON [0.50];
         ApDiscSwi               : OFF [0.50], ON [0.50];
         ApEngSwi                : OFF [0.50], ON [0.50];
         ApprSwi                 : OFF [0.50], ON [0.50];
         FdSwi                   : OFF [0.50], ON [0.50];
         FlcSwi                  : OFF [0.50], ON [0.50];
         GaSwi                   : OFF [0.50], ON [0.50];
         HdgSwi                  : OFF [0.50], ON [0.50];
         LapprTrackCondMet       : FALSE [0.50], TRUE [0.50];
         NavSwi                  : OFF [0.50], ON [0.50];
         NavTrackCondMet         : FALSE [0.50], TRUE [0.50];
         Overspeed               : FALSE [0.50], TRUE [0.50];
         SyncSwi                 : OFF [0.50], ON [0.50];
         TransSwi                : OFF [0.50], ON [0.50];
```

```
        VapprTrackCondMet       : FALSE [0.50], TRUE [0.50];
        VsPthWhlMot             : FALSE [0.50], TRUE [0.50];
        VsSwi                   : OFF [0.50], ON [0.50]

    Other_Input :
       DEFAULT :
        AltSel      : FALSE [0.50], TRUE [0.50];
        AltselAct   : FALSE [0.50], TRUE [0.50];
        AltselSel   : FALSE [0.50], TRUE [0.50];
        AltselTrk   : FALSE [0.50], TRUE [0.50];
        FdOn        : FALSE [0.50], TRUE [0.50];
        FGSActive   : FALSE [0.50], TRUE [0.50];
        FlcSel      : FALSE [0.50], TRUE [0.50];
        HdgSel      : FALSE [0.50], TRUE [0.50];
        LapprAct    : FALSE [0.50], TRUE [0.50];
        LapprSel    : FALSE [0.50], TRUE [0.50];
        LgaSel      : FALSE [0.50], TRUE [0.50];
        ModesOn     : FALSE [0.50], TRUE [0.50];
        NavAct      : FALSE [0.50], TRUE [0.50];
        NavSel      : FALSE [0.50], TRUE [0.50];
        PthSel      : FALSE [0.50], TRUE [0.50];
        RollSel     : FALSE [0.50], TRUE [0.50];
        VapprAct    : FALSE [0.50], TRUE [0.50];
        VapprSel    : FALSE [0.50], TRUE [0.50];
        VgaSel      : FALSE [0.50], TRUE [0.50];
        VsSel       : FALSE [0.50], TRUE [0.50]
END MESSAGE_PROBABILITY
```

Appendix B

Statistical Profile Syntax

The following definitions have been added to the Requirements State Machine Language without Events ($RSML^{-e}$) specification language in order to permit statistical testing using the specification as an operational profile.

```
def:                        : type_def          /* Existing */
                                :                     :
                            | message_def       /* Existing */
                            | interface_prob_table_def
                            | message_prob_table_def
                            ;

/*---------- Interface Probability Table --------------------------*/
interface_prob_table_def : INTERFACE_PROBABILITY
                             intf_prob_case_def
                           END INTERFACE_PROBABILITY
                         ;

intf_prob_case_def       : /* EMPTY */
                         | intf_prob_case_def intf_prob_case
                         | intf_prob_case_def default_intf_prob_case
                         ;

intf_prob_case           : CONDITION ':' condition
                             intf_prob_def
                         ;

default_intf_prob_case   : DEFAULT ':'
                             intf_prob_def
                         ;

intf_prob_def            : interface_prob ';'
                         | intf_prob_def interface_prob ';'
                         ;

interface_prob           : IDENTIFIER '[' REAL_VALUE ']'
```

```
                    ;

/*---------- Message Probability Table ----------------------------*/

message_prob_table_def  : MESSAGE_PROBABILITY
                            msg_fld_intf_case_def
                          END MESSAGE_PROBABILITY
                        ;

msg_fld_intf_case_def   : msg_fld_intf_case
                        | msg_fld_intf_case_def msg_fld_intf_case
                        ;

msg_fld_intf_case       : IDENTIFIER ':'
                            msg_fld_prob_case_def
                        ;

msg_fld_prob_case_def   : /* EMPTY */
                        | msg_fld_prob_case_def msg_fld_prob_case
                        | msg_fld_prob_case_def def_msg_fld_prob_case
                        ;

msg_fld_prob_case       : CONDITION ':' condition
                            msg_fld_prob_phrase_def
                        ;

def_msg_fld_prob_case   : DEFAULT ':'
                            msg_fld_prob_phrase_def
                        ;

msg_fld_prob_phrase_def : msg_fld_prob_phrase
                        | msg_fld_prob_phrase_def ';' msg_fld_prob_phrase
                        ;

msg_fld_prob_phrase     : IDENTIFIER ':' msg_fld_probability
                        ;

msg_fld_probability     : IDENTIFIER '[' REAL_VALUE ']'
                        | IDENTIFIER '[' REAL_VALUE ']' ',' msg_fld_probability
                        | INT_VALUE '[' REAL_VALUE ']'
                        | INT_VALUE '[' REAL_VALUE ']' ',' msg_fld_probability
                        | TRUE_TOKEN '[' REAL_VALUE ']'
                        | TRUE_TOKEN '[' REAL_VALUE ']' ',' msg_fld_probability
                        | FALSE_TOKEN '[' REAL_VALUE ']'
                        | FALSE_TOKEN '[' REAL_VALUE ']' ',' msg_fld_probability
                         ;
```

Appendix C

Statistical Test Case Generator Design Document

C.1 System Purpose

This system will statistically generate test cases based on an operational profile. The operational profile will be implemented as a state-based requirements specification model augmented with conditional probabilities on system inputs. This design document details the existing system and the changes and additions required for the new functionality.

C.2 Architecture

This system will be built within and in conjunction with the NIMBUS software development environment. The NIMBUS system provides simulation, prototyping, model checking, and structural test case generation capabilities for state-based specification models of computer systems. The models are composed using the Requirement State Machine Language without Events (RSML^{-e}).

Inputs/Outputs

NIMBUS takes an RSML^{-e} specification model as input. It can also be set up to receive information from the environment using a channels and interfaces concept.

Nimbus Simulator Subsystems

The NIMBUS simulator (NimbusSim) is made up of six primary subsystems, as shown in Figure C.1. The graphical user interface invokes the command framework to execute commands of the NIMBUS environment. The command framework uses the parser to generate the abstract syntax tree representation of the system model and can call the simulation engine to run simulations of the specification model. The observer framework is used to monitor the simulation and report changes of interest to the graphical user interface.

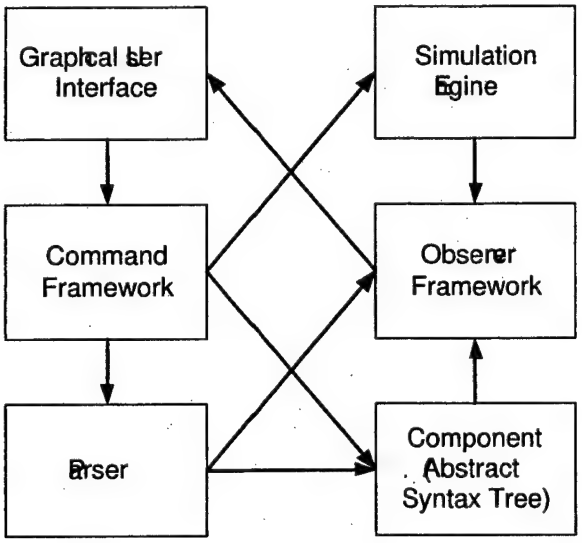

Figure C.1: Nimbus Simulator Subsystems.

This design calls for additional commands to be inserted into the NIMBUS command framework to provide invocation of the new components. The parser will also require updates to address the new structures. These new components are described in more detail in the following section and in appendices. All the described changes and additions to NIMBUS are based on the version of NIMBUS as of 15 January 2003 (version 1.3.0.1). When the implementation of this design is completed, the changes

will be considered for addition to the distribution version of NIMBUS.

C.3 Statistical Test Case Generation (stattest)

The test generation component will generate test cases statistically, using an operational profile. This component will take the structural model in the abstract syntax tree (AST) and the statistical profile and generate test cases, which will consist of a series of messages to be given to the system under test. It will be called from the NimbusSim command framework, with a specification model loaded and in its initial configuration. In each active state, this component will determine valid interfaces and choose one to be the active interface, based on the statistical profile. It will also determine the values to be given to the fields in the message associated with the active interface. These will be recorded in the test case, along with the resulting next state values, for oracle purposes.

Input/Output

This component has two basic requirements, a state-based specification model and a statistical profile must be loaded in the Nimbus simulator. These will comprise the two components of an operational profile; the model provides the structural component and the statistical profile provides the statistical component. The Nimbus parser must be augmented to address the new structures found in the statistical profile. The new components are defined in Section C.4 and the changes to the parser are addressed in Section B. The signatures of these function calls from the command framework are as follows:

`stattest -start`	start test case generation (output to cout)
`stattest -start filename`	start test case generation (output to filename)
`stattest -stop`	stop test case generation (returns output to cout)
`stattest -write`	write out probability tables
`statrun integer`	run `integer` steps
`statrun integer1 integer2`	run `integer2` test cases of `integer1` steps each

where `filename` is the test case file name. An example, once the model is loaded into NimbusSim, the test case generation machinery is started using `stattest -start` or `stattest -start filename`. Then a single test case can be run using the first `statrun` command format or a whole test suite can be generated using the second `statrun` command format. The `stattest -stop` command format closes out all test case generation and returns output to standard out, if necessary.

Actions

This function will be invoked after a model has been loaded into the Nimbus simulator and verified as complete and correct. Once the function has been invoked, the following steps will take place, as shown in Figure C.2:

1. Initialize test case counter, *tcc*, to 1.
2. Open new test case, named `<filename>`
3. Request state of model from NimbusSim.
4. Record the initial state of the model from NimbusSim in the test case.
5. Initialize message counter, *mc*, to 1.
6. Generate a message from current state. (Within stattest, using the AST)
7. Record the message in the test case.
8. Send message to NimbusSim.

9. Record the resulting state of the model in the test case.
10. Increment message counter. ($mc = mc + 1$)
11. If $mc <= tclen$, repeat from Step 6.
12. Close test case.
13. Increment test case counter. ($tcc = tcc + 1$)
14. If $tcc <= tcnum$, repeat from Step 2.
15. Test cases complete, return to command line.

Notes

The changes to existing classes and new classes required for these new components are described in detail below and the additions to the system grammar are found in Appendix B.

C.4 Probability Table Classes

There are several new classes that must be added to the abstract system tree in order to attach probabilities to the model and conduct statistical testing processes. The two primary classes are RInterfaceProbabilityTable, which defines the interface probability table, and RMessageProbabilityTable, which defines the message probability table. The remainder of the new classes are in support of these two classes.

Interface Probability Table

Figure C.3 shows the UML Diagram for the Interface Probability Table.

Message Field Probability Table

Figure C.4 shows the UML Class diagram for the Message Field Probability Table.

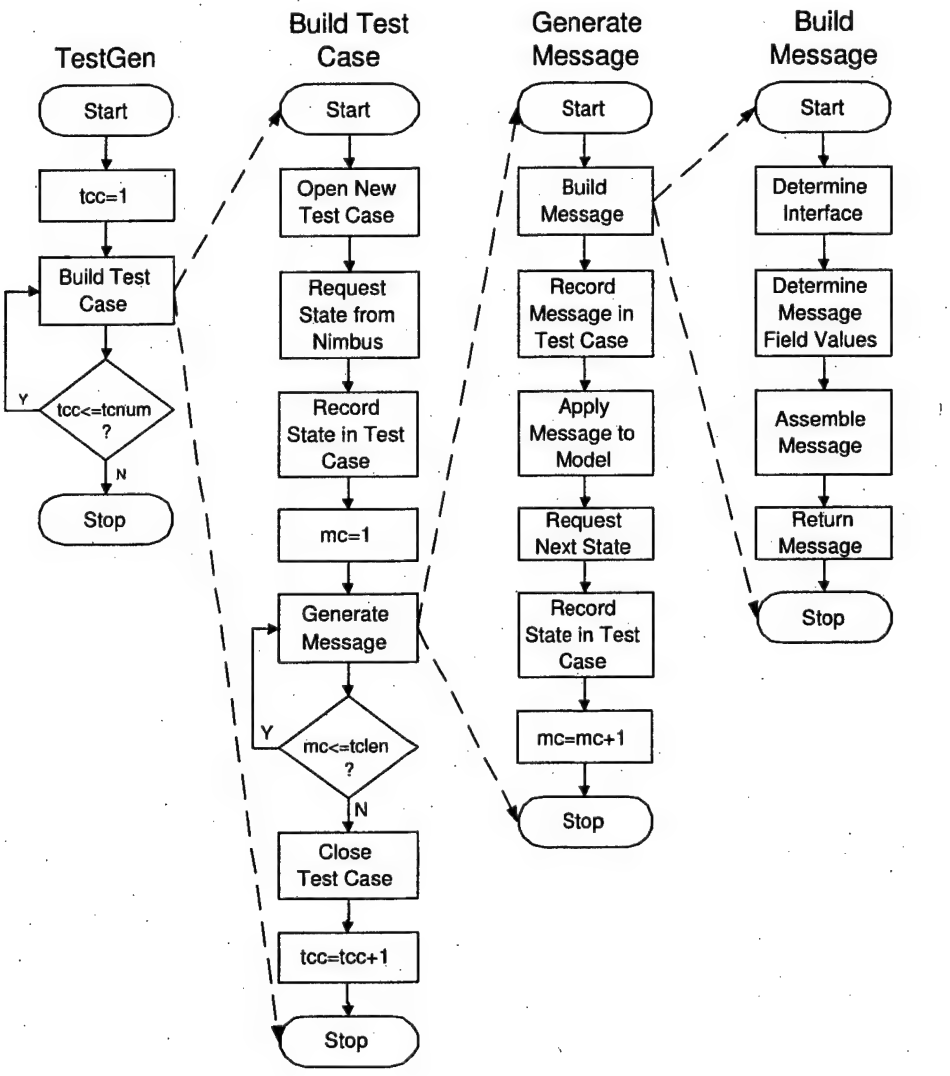

Figure C.2: Flowchart for Statistical Testing Process.

New Type Definitions

The six lists shown above are implemented as vectors, as illustrated below:

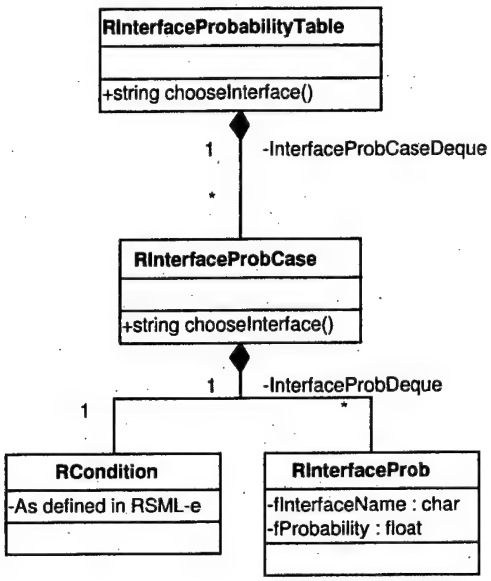

Figure C.3: UML Diagram of the Interface Probability Table.

```
typedef std::vector<RInterfaceProbCase*>    InterfaceProbCaseDeque;
typedef std::vector<RInterfaceProb*>        InterfaceProbDeque;
typedef std::vector<RMsgFieldIntCase*>      MsgFieldIntCaseDeque;
typedef std::vector<RMsgFieldProbCase*>     MsgFieldProbCaseDeque;
typedef std::vector<RMsgFieldProbPhrase*>   MsgFieldProbDeque;
typedef std::vector<RMsgFieldProb*>         MsgFieldProbDeque;
```

The vector provides easy insertion of new items during parsing of the operational profile, yet retains the ordering of the items, which will be of importance when generating test cases.

C.5 New Class Definitions

RInterfaceProbabilityTable

RInterfaceProbabilityTable contains a set of RInterfaceProbCases and is the representation of the interface probability table. It will be implemented as a data com-

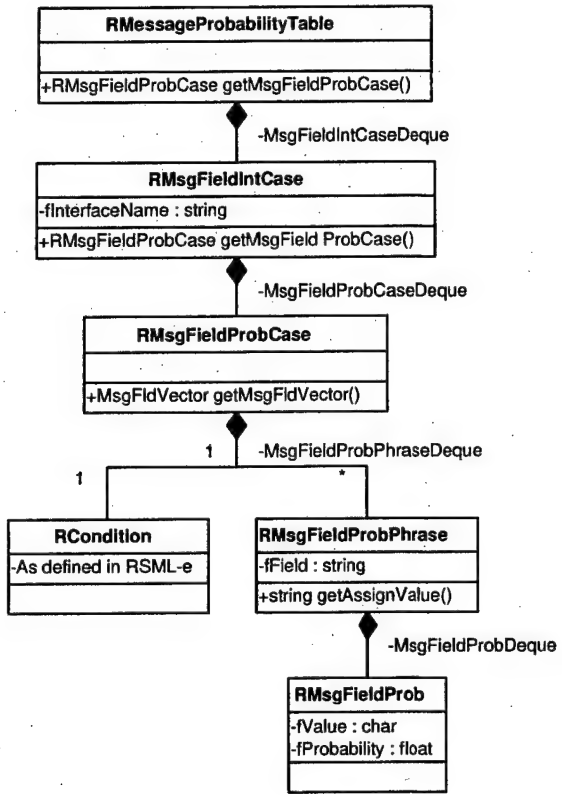

Figure C.4: UML Diagram for Message Field Probability Table.

ponent of RComponent. Its primary method is chooseInterface(double), which uses a supplied random number to choose an interface to consider active. It returns the interface name as a string. It picks the interface by iterating through the InterfaceProbCaseDeque until it finds one whose condition is evaluates to true, then it calls its chooseInterface() method.

```
class RInterfaceProbabilityTable : public RDataStructureObject
public:
  RInterfaceProbabilityTable(std::string name="");
  virtual ~RInterfaceProbabilityTable(){}
// set the data elements
  void setName(std::string name);
  void insertInterfaceProbCase(const RInterfaceProbCase *) {
```

```
        fInterfaceProbCaseDeque->push_back( ipc );}
//get the data elements
  virtual std::string name() const;
  std::string chooseInterface( double ) const {
     InterfaceProbCaseDeque::const_iterator ipcd_itr;
     for (ipcd_itr = fInterfaceProbCaseDeque->begin();
          ipcd_itr != fInterfaceProbCaseDeque->end(); ipcd_itr++)
        if ((*ipcd_itr).condition())
           return(*ipcd_itr).chooseInterface(number);
}
private:
   InterfaceProbCaseDeque* fInterfaceProbCaseDeque;
```

RInterfaceProbCase

RInterfaceProbCase consists of a condition and a deque of RInterfaceProbs. Its chooseInterface method iterates through the InterfaceProbDeque and summing the probabilities until a threshold value is reached or exceeded. It then returns the interfaces name as a string value.

```
class RInterfaceProbCase : public RDataStructureObject
public:
  RInterfaceProbCase(RCondition *cond=NULL, InterfaceProbDeque *ipd=NULL);
  virtual ~RInterfaceProbCase(){}

//set the data elements
  void insertInterfaceProb(RInterfaceProb &intProb)
          { fInterfaceProbDeque->push_back(intProb); }
//get the data elements
  virtual std::string name() const;
  bool condition() const {return fCondition->value();}
  RInterfaceProb *interfaceProb( std::string ) const {
     InterfaceProbDeque::const_iterator ipd_itr;
     for (ipd_itr=fInterfaceProbDeque->begin();
          ipd_itr != fInterfaceProbDeque->end(); ipd_itr++)
        if ((*ipd_itr).interfaceName() == interf)  return (*ipd_itr);
  }
```

```
  std::string chooseInterface( double number ) const {
     double z = 0.0;
     InterfaceProbDeque::const_iterator ipd_itr;
     for (ipd_itr=fInterfaceProbDeque->begin();
          ipd_itr != fInterfaceProbDeque->end(); ipd_itr++)
        { z += (*ipd_itr).probability();
          if ( z >= number )  return (*ipd_itr).interfaceName();
        }
  }
private:
    RCondition*         fCondition;
    InterfaceProbDeque* fInterfaceProbDeque;
```

RInterfaceProb

Holds the interface name and probability pairs for the methods described above.

```
class RInterfaceProb : public RDataStructureObject
public:
  RInterfaceProb(std::string interf, double prob);
  virtual ~RInterfaceProb(){}
//set the data elements
  void setInterfaceName(std::string interf);
  void setProbability(double prob);
//get the data elements
  virtual std::string name()  const;
  std::string interfaceName() const;
  double probability() const;
private:
    std::string fInterfaceName;
    double      fProbability;
```

RMessageProbabilityTable

Holds the message probability table. getMsgFieldProbCase(string)is its primary function, which iterates through the fMsgFieldIntCaseDeque until it encounters the Msg-

FieldIntCase whose name matches the string passed in. Then it calls that MsgFieldIntCase's getMsgFieldProbCase() method.

```
class RMessageProbabilityTable : public RDataStructureObject
public:
  RMessageProbabilityTable(std::string name="");
  virtual ~RMessageProbabilityTable(){}

  void setName(std::string name);
    void *insertMsgFieldIntCase( RMsgFieldIntCase *){
       fMsgFieldIntCaseDeque->push_back(msgFieldIntCase);
    }
  virtual std::string name() const;
  RMsgFieldIntCase  *getMsgFieldIntCase(const std::string ) const {
     MsgFieldIntCaseDeque::const_iterator mficd_itr;
     for(mficd_itr=fMsgFieldIntCaseDeque->begin();
         mficd_itr!=fMsgFieldIntCaseDeque->end(); mficd_itr++)
        if ((*mficd_itr).name() == name) return (*mficd_itr);
  }
  RMsgFieldProbCase *getMsgFieldProbCase( std::string ) const {
     MsgFieldIntCaseDeque::const\_iterator mficd_itr;
     for(mficd_itr=fMsgFieldIntCaseDeque->begin();
         mficd_itr!=fMsgFieldIntCaseDeque->end(); mficd_itr++)
        if ((*mficd_itr).name() == name)
           return(*mficd_itr).getMsgFieldProbCase();
  }
private:
    MsgFieldIntCaseDeque*   fMsgFieldIntCaseDeque;
```

RMsgFieldIntCase

The RMsgFieldIntCase class holds all the interface names and their associated conditional probability cases. Its primary method, getMsgFieldProbCase(), iterates through the current deque until it finds one RMsgFieldProbCase object whose condition evaluates to true. It then returns that object.

```
class RMsgFieldIntCase : public RDataStructureObject
```

```
public:
  RMsgFieldIntCase(std::string interf, MsgFieldProbCaseDeque *mfpcd);
  virtual ~RMsgFieldIntCase(){}
//set the data elements
  void setInterfaceName(std::string name);
  void insertMsgFieldProbCase(const RMsgFieldProbCase &msgFieldProbCase )
     { fMsgFieldProbCaseDeque->push_back(msgFieldProbCase); }
//get the data elements
  virtual std::string name() const;
  RMsgFieldProbCase *getMsgFieldProbCase() const {
     MsgFieldProbCaseDeque::const_iterator mfpcd_itr;
     for (mfpcd_itr=fMsgFieldProbCaseDeque->begin();
          mfpcd_itr!=fMsgFieldProbCaseDeque->end(); mfpcd_itr++)
        if ((*mfpcd_itr).condition() ) return (*mfpcd_itr);
  }
private:
  std::string              fInterfaceName;
  MsgFieldProbCaseDeque*   fMsgFieldProbCaseDeque;
```

RMsgFieldProbCase

This class holds a condition and the probability assignment phrases that go with it. Its primary method is getMsgVector(), which iterates through the message field probability phrases and constructs a vector of message field values for constructing a message. It calls the probability assignment phrase's getAssignValue() method.

```
class RMsgFieldProbCase : public RDataStructureObject
public:
  RMsgFieldProbCase(RCondition *cond, MsgFieldProbPhraseDeque *mfppd) ;
  virtual ~RMsgFieldProbCase(){}
//set the data elements
  void insertMsgFieldProbPhrase(const RMsgFieldProbPhrase &mfpp)
     { fMsgFieldProbPhraseDeque->push_back(mfpp); }
//get the data elements
  virtual std::string name() const;
  bool condition() const;
  RMsgFieldProbPhrase *msgFieldProbPhrase(std::string var) const {
```

```
    MsgFieldProbPhraseDeque::const_iterator mfppd_itr;
    for (mfppd_itr=fMsgFieldProbPhraseDeque->begin();
         mfppd_itr!=fMsgFieldProbPhraseDeque->end(); mfppd_itr++)
       if ((*mfppd_itr).varName() == var) return (*mfppd_itr);
  }
private:
  RCondition*              fCondition;
  MsgFieldProbPhraseDeque* fMsgFieldProbPhraseDeque;
```

RMsgFieldProbPhrase

This class holds the name of a variable to be assigned and the value-probability pairs needed for the operational profile. Its getAssignValue() method iterates through the choices, selecting a value when the probability sum meets or exceeds randomly generated value.

```
class RMsgFieldProbPhrase : public RDataStructureObject
public:
  RMsgFieldProbPhrase const (std::string varName,  RMsgFieldProb &mfp);
  virtual ~RMsgFieldProbPhrase(){}
//set the data elements
  void setVariableName(std::string varName);
  void insertMsgFieldProb (const RMsgFieldProb &probPair) const
     {fMsgFieldProbDeque->push_back(probPair); }
//get the data elements
  virtual std::string name() const;
  std::string varName() const;
  RMsgFieldProb *msgFieldProb(std::string value) const {
     MsgFieldProbDeque::const_iterator mfpd_itr;
     for (mfpd_itr=fMsgFieldProbDeque->begin();
          mfpd_itr!=fMsgFieldProbDeque->end(); mfpd_itr++)
        if ((*mfpd_itr).value() == value) return (*mfpd_itr);
  }
private:
  std::string        fVariableName;
  MsgFieldProbDeque* fMsgFieldProbDeque;
```

RMsgFieldProb

This class holds the actual message field value-probability pairs for the operational profile.

```
class RMsgFieldProb : public RDataStructureObject {
public:
  RMsgFieldProb(std::string val, double prob);
  virtual ~RMsgFieldProb() {}
//set the data elements
  void setValue(std::string value);
  void setProbability(double prob);
//get the data elements
  virtual std::string name() const;
  std::string value() const;
  double probability() const;
private:
  std::string fValue;
  double      fProbability;
```

C.6 Changes to Existing Classes

RComponent

RComponent needs to have two additions to its structure, the addition of the interface probability table and the message probability table, complete with their get and set functions. This will require type definitions for RInterfaceProbabilityTable and RMessageProbabilityTable.

```
// get methods
RProbabilityTable &ProbabilityTable ();

// set methods
void setProbabilityTable (RProbabilityTable &probabilityTable);

RProbabilityTable  fInterfaceProbabilityTable;
```

Appendix D

Statistical Test Case Generator User's Guide

The statistical testing components were designed to be a straightforward, easy-to-use add-on to the NIMBUS environment. The primary command, `statsuite`, is a compilation of several other commands; statistical testing, therefore, can be performed automatically or manually for more flexibility and control. This section describes the automatic `statsuite` command first, then each of the commands that make up this statistical testing suite command.

D.1 `statsuite`

The `statsuite` command is designed to be run in the NIMBUS simulator with an RSML^{-e} model loaded that includes a valid operational profile, as defined in Appendix A. The `statsuite` command generates test cases, applies them to the identified specifications under test, and records the state information from those tests. The only portion of testing it does not perform is to compare the oracle results with the actual results, but it does create the necessary DOS batch files to perform these comparisons.

The usage of `statsuite` is as follows:

```
statsuite <#steps><#testCases><#specs><filename>
```

In this command, the user specifies the number of steps for each test case, the number of test cases to generate, the number of specifications being tested, and the base of

the filename used for the specifications. The filename base is designed to allow the testing of multiple specifications with the same test cases. For example, the command

```
statsuite 10 20 5 Spec
```

would generate twenty test cases of ten steps each and then apply those test cases to five specifications, named `Spec0.nimbus`, `Spec1.nimbus`, `Spec2.nimbus`, `Spec3.nimbus`, and `Spec4.nimbus`. The state information results would be recorded to output files for later analysis. This command also generates a DOS batch file, `Spec.bat`, which can be run to perform the comparisons of the actual results and the results expected from the oracle specification. It will compare the results for all five specifications as noted in the command.

Each of the steps in this `statsuite` command is presented below in further detail.

D.2 `stattest`

The `stattest` command has three forms,

```
stattest -write,
stattest -run <#steps>, and
stattest -run <#steps> <#testCases>.
```

The `stattest -write` command is used to output a copy of the operational profile's probability tables. This is primarily helpful in checking that the tables are written correctly in the operational profile.

The `stattest -run` version of the command has the two latter forms. The first creates a single test case with the number of steps identified. The second allows the user to also specify the number of test cases to generate. This is the version used by the `statsuite` command to generate the test suite. The results of this command, the input messages and the resulting state information, are written to a file in the ITR format as identified in Section 4.2.1. The file is named with the prefix of the oracle

specification file but with a `.itr` extension.

D.3 statparse

Once the test suite is in the ITR format, we can use the `statparse` command to write out the input message files and the oracle results files needed for testing and analysis. The command is used as follows:

```
statparse <filename>.itr
```

where filename.itr is the name of the ITR file. This command was beneficial to our testing, as the test suites generated by SMV for structural testing were also stored in this format. We ran this command and produced test input files for the NIMBUS simulator as well as SMV.

D.4 statscript

The `statscript` command produces scripts for three functions, (1) executing the test suite on the target specification, (2) comparing the actual results with the oracle results, and (3) measuring the coverage of a model by the test suite. The command is used as follows:

```
statscript <filename> <#specs> <#testCases>
```

where filename is again the base of the filenames used for the specifications under test and the number of specifications and test cases must be provided. In the `statsuite` command, this is all handled internally.

The resulting script files defined above are named as, (1) `filename.nscript`, (2) `filename.bat`, and (3) `oraclefilenamecover.nscript`, where oraclefilename is the name of the file used to produce the test suite. In our testing, we used `FGS05.nimbus` as the oracle and `ToyFGS05_Left_VarReplaceFault` as our specification under test

base filename. With `FGS05.nimbus` loaded into the simulator, running

```
statscript ToyFGS05_Left_VarReplaceFault 5 10
```

produces the three script files,

`ToyFGS05_Left_VarReplaceFault.nscript`,

`ToyFGS05_Left_VarReplaceFault.bat`, and

`FGS05cover.nscript`.

Only the first of these scripts is used by the `statsuite` command. The second, the DOS batch file, must be run from a command prompt window or from Windows Explorer and the last script must be run separately in NIMBUS with the `coverage` command, discussed below.

D.5 script

The `script` command was not created for the statistical testing add-on to NIMBUS. It is a preexisting command and is used to apply the test cases to the specification under test. The form of the command,

```
script -r <scriptname>
```

indicates to read (-r) the script found in the file "scriptname." This file is the one generated above by the `statscript` command. `Statsuite` uses this command to automatically execute the test suite it has generated and parsed.

D.6 coverage

The `coverage` command was created within our research group to monitor the state of the model during testing and measure the state, transition, and transition decision coverage provided by a given test suite. This command sets up the necessary observers, then processes the script given in the command.

```
coverage FGS05cover.nscript
```

runs the script produced in the `statscript` command outlined above. It applies the test suite to the model loaded in the simulator and outputs the three metrics to the console. This is not used in the `statsuite` command, as it is not essential to the testing and analysis we perform.